Methods and Tools for Mapping Process Networks onto Multi-Processor Systems-On-Chip

Iuliana Bacivarov, Wolfgang Haid, Kai Huang, Lothar Thiele

Abstract Applications based on the Kahn process network (KPN) model of computation are determinate, modular, and based on FIFO communication for inter-process communication. While these properties allow KPN applications to efficiently execute on multi-processor systems-on-chip (MPSoC), they also enable the automation of the design process. This chapter focuses on the second aspect and gives an overview of methods for automating the design process of KPN applications implemented on MPSoCs. Whereas previous chapters mainly introduced techniques that apply to restricted classes of process networks, this overview will be dealing with general Kahn process networks.

1 Introduction

Multi-processor system-on-chip (MPSoC) is one of the most promising and solid paradigm for implementing embedded systems for signal processing in communication, medical, and multi-media applications. MPSoC platforms are heterogeneous by nature as they use multiple computation, communication, memory, and peripheral resources. They allow the parallel execution of (multiple) applications and, at the same time, they offer the flexibility to optimize performance, energy consumption, or cost of the system. Nevertheless, to optimize an MPSoC in the presence of tight time-to-market and budget constraints, a systematic design flow is required.

To deal with this challenge, Kienhuis *et al.* [1] suggested to structure the design flow in a certain manner, now commonly referred to as the Y-chart approach. It is a systematic methodology for selecting an embedded system implementation from a set of alternatives, a process often denoted as design space exploration. One key idea underlying this approach is to explicitly separate application and architecture

Iuliana Bacivarov, Wolfgang Haid, Kai Huang, Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland
e-mail: firstname.lastname@tik.ee.ethz.ch

specifications. A separate mapping specification describes how the application is spatially (binding) and temporally (scheduling) executed on the architecture. Design space exploration is then performed by iteratively analyzing and optimizing the application, the structure of the underlying (hardware) architecture as well as candidate mappings, as shown in Fig. 1.

Many design flows implementing the Y-chart approach have been proposed. For a review, see [2]. These flows have in common that they impose a set of system-level concepts to facilitate design space exploration, such as the use of a formal model of computation, providing restrictions on the set of scheduling policies, and relying on modular specifications. For instance, the application may be formally specified as a data flow model, a synchronous model, or a discrete event model, in order to enable automated performance analysis. In a similar way, resource sharing policies may be limited to an event-triggered or a time-triggered policy, to prune the design space. Finally, using modular system-level specifications will enable quick system modifications concerning the application, architecture, and mapping.

In the context of (array) signal processing applications executing on MPSoC, the Kahn process network (KPN) model of computation [3] is frequently used. Assuming a network of autonomous, concurrently executing processes that communicate point-to-point via unbounded FIFO channels, the KPN model has additional favorable properties. The KPN model is determinate, i.e. the functional behavior is independent on the scheduling of processes. The inter-process communication via FIFO channels using blocking read semantics can be efficiently implemented either in software, hardware, or in heterogeneous HW/SW systems. Computation and control are completely distributed, requiring no global synchronization, communication, or memory. The resulting modularity allows applications to be scaled easily and opens up many degrees of freedom for implementing a system.

Due to these properties, the KPN model of computation is “compatible” with the Y-chart approach and has led to numerous design flows. Although they share the same model of computation, these design flows consider different design objectives, they focus on different aspects, and leverage different properties of the KPN model

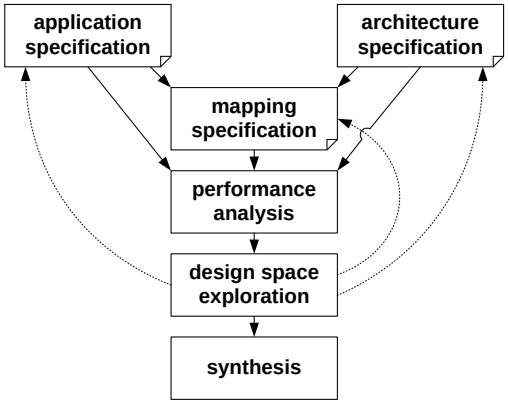


Fig. 1 Y-chart approach for designing MPSoC.

or one of its subclasses. In this chapter, an overview of KPN-based design flows is given, emphasizing both, similarities and differences in these flows. The following section reviews existing design flows and the way they relate to the Y-chart. Afterwards, a closer look at individual steps in the design flow is taken and several methods to tackle them are presented. Finally, for exemplification, a specific design flow is considered in detail.

2 KPN Design Flows for Multiprocessor Systems

Several design flows based on the Y-chart approach and the KPN model have been developed. Table 1 shows a (non-exhaustive) list of design flows targeted at the implementation of KPN applications on MPSoC platforms. In addition to the listed design flows, there are other approaches related to the KPN model with different aims. Ptolemy [4] and Metropolis [5] allow the analysis and simulation of applications specified as KPNs, among other models of computation. However, they are targeted more towards hardware/software codesign and in particular towards the system synthesis and verification. The design space exploration is not the main focus of these frameworks. The Mathworks Real-Time Workshop [6] and the National Instruments LabVIEW Microprocessor SDK [7] target the implementation of signal processing applications on single-processor systems. SystemCoDesigner [8] and PeaCE [9] are HW/SW codesign flows based on a model of computation that combines the KPN model with finite state machines, see chapter [10]. Note that even though the focus of this chapter is on the design flows for MPSoC listed in Table 1, many of the presented ideas also apply to the other mentioned design flows.

Table 1 KPN design flows for MPSoCs.

design flow	**web page**
Artemis [11]	http://daedalus.liacs.nl
Distributed Operation Layer (DOL) [12]	http://www.tik.ee.ethz.ch/∼shapes/dol.html
Embedded System-Level Platform Synthesis and Application Mapping (ESPAM) [13]	http://daedalus.liacs.nl
Koski [14]	not available online
Multiapplication and Multiprocessor Synthesis (MAMPS) [15]	http://www.es.ele.tue.nl/mamps
Open Dataflow (OpenDF) [16]	http://opendf.sourceforge.net
Software/Hardware Integration Medium (SHIM) [17]	not available online
StreamIt [18]	http://www.cag.lcs.mit.edu/streamit

Generally, KPN design flows for MPSoCs respect the four design phases of the Y-chart: system specification, performance analysis, design space exploration, and system synthesis, as shown in Fig. 1.

Based on these four phases, the design process can be described as follows: The starting point of the design flow is a parallelized KPN specification of the application. In this specification, the coarse-grain data and functional parallelism of the application is made explicit. Fine-grained word or instruction-level parallelism can effectively be handled by today's compilers. Usually, the KPN is manually specified by the programmer. There are, however, also tools available that allow deriving a KPN from sequential programs, such as the Compaan [19] and pn [20] tools. KPN design flows usually provide a functional simulation capability that enables the execution of KPN specification on a standard single-processor machine in a multitasking environment. Due to the determinacy of KPNs, the timing-independent functionality of the application can be validated this way.

Second, the architecture needs to be specified. This is frequently done in form of a system-level specification describing the architectural resources, such as processors, memories, interconnects, and I/O devices. This specification can either describe a fixed MPSoC or the template of a configurable MPSoC platform. In both cases, the architecture specification needs to contain all the information required for design space exploration and performance analysis. In the case of a configurable platform, the architecture specification is also the basis for the synthesis of the final target platform later in the design flow. Hence, it needs to contain information required by the RTL synthesis tool, such as references to VHDL or Verilog code of hardware components, complete IP blocks, and configuration files.

The application and architecture specification phase is followed by defining a mapping of the application onto the architecture. In this step, processes are bound to processors and channels are bound to communication paths containing memories and interconnects. In addition, the scheduling and arbitration policies for shared resources are defined.

Usually, the final mapping is the result of a design space exploration, which is done based on the system performance analysis. The methods applied for performance analysis range from simple back-of-the-envelope calculations to formal analysis methods, simulations, and measurements. In KPN design flows, performance analysis during design space exploration is possible and is usually done at a rather high level of abstraction. As shown in the next section, different methods targeted towards KPN applications have been proposed in this context that achieve high accuracy within short analysis times. Being able to defer the use of simulation or measurements until late in the design cycle is one of the key advantages of KPN design flows.

After manual or automated design space exploration, the system is finally implemented by making use of appropriate synthesis techniques. For this purpose, KPN design flows feature powerful synthesis tools that implement a system based on the application, architecture, and mapping specification in software, hardware, or both hardware and software. Clearly, this is a key advantage of KPN design flows because the pitfalls of implementing a parallel system, such as hardware-software interface generation, deadlocks, starvation, and data races are handled in an automated way.

The design flows listed in Table 1 implement this basic Y-chart approach in different ways: On the one hand, the methods that are applied in each of the four phases

differ between the design flows, as discussed in the next section. On the other hand, the scope (set of optimization variables) of design space exploration is different. Basically, one can distinguish between software design flows, where the target platform is fixed, and hardware/software co-design flows, where a template of a target platform is given and the instantiation of a specific platform is part of the design space exploration. This is shown in Table 2 where a few case studies are summarized that have been performed using the design flows listed in Table 1. DOL, SHIM, and StreamIt assume fixed hardware platforms, whereas the scope of the other design flows encompasses the implementation of the target platform on FPGAs.

Table 2 Please write your table caption here

design flow	**case study application**	**target platform**	**performance analysis**	**exploration method**
Artemis [21]	motion-JPEG encoder	Molen architecture on Xilinx Virtex-II Pro FPGA	trace-driven simulation	evolutionary algorithm
DOL [12]	MPEG-2 decoder	Atmel DIOPSIS 940	real-time analytic model	evolutionary algorithm
ESPAM [13]	motion-JPEG encoder	multi-MicroBlaze on Xilinx Virtex-II Pro FPGA	measurement	exhaustive search
Koski [14]	WLAN terminal	multi-NIOS on Altera Stratix-II FPGA	high-level simulation	simulated annealing
MAMPS [15]	H263 and JPEG decoders	multi-MicroBlaze on Xilinx Virtex-II Pro FPGA	high-level simulation	dedicated heuristic
OpenDF [22]	MPEG-4 SP decoder	FPGA (no particular type specified)	not applicable	not applicable
SHIM [23]	JPEG decoder	Sony/Toshiba/IBM Cell BE	not applicable	not applicable
StreamIt [18]	12 streaming applications	RAW architecture	SDF analytic model	simulated annealing

3 Methods

KPN design flows attempt to assist a system designer in implementing an application as a hardware/software system by offering support for several activities such as:

- system specification,
- system synthesis,
- performance analysis, and
- design space exploration.

For each of these activities, methods have been proposed that differ in goal, scope, degree of automation, and complexity. In the previous chapters, mainly methods for subclasses of KPNs have been discussed. In this chapter, we give an overview of

methods that are applicable to general KPNs in the context of MPSoCs. For each of the activities mentioned above, we discuss the challenges and proposed solutions.

3.1 System Specification

Developing applications that run correctly and efficiently on MPSoCs is challenging. The difficulty consists in finding an appropriate level of abstraction that balances the conflicting goals of (a) developing applications in a productive manner and of (b) enabling efficient automated implementation. While productivity is usually achieved by programming at a high abstraction level, efficiency is usually achieved by optimizing code at a low abstraction level. Many case studies provide evidence that for streaming applications, the KPN model of computation achieves a good trade-off between these two goals. On the one hand, streaming applications can often naturally be modeled as a KPN which promotes productivity. On the other hand, runtime environments have been developed that efficiently implement processes and channels.

Specifically, the KPN model can be seen as a *coordination* model [24] which considers the programming of a distributed system as the combination of two distinct activities: the actual *computing* part comprising a number of processes involved in manipulating data and a *coordination* part reflecting the communication and cooperation between processes. The coordination model allows reuse of components because the application programmer can easily build new algorithms by a new composition of existing processes. Furthermore the coordination model allows applications to be ported to different target architectures because usually only the glue-code that implements the coordination part is architecture dependent.

Due to these reasons, KPN applications are usually specified in a way that reflects the coordination model. Two different approaches can be distinguished, namely specification using a host as well as a coordination language, and specification using a domain-specific language. When using distinguished host and coordination languages, the KPN processes are specified in a host language (often in C or C++) whereas the coordination part is specified separately using a coordination language (often in XML or UML). This is, for instance, the approach taken in the Artemis and DOL design flows, where C and XML are used. When using a domain-specific language, computation and coordination are expressed in a single language that provides constructs for both parts. OpenDF and StreamIt, for instance, are based on domain-specific languages.

In both cases, applications are usually expressed based on the principles of encapsulation and explicit concurrency: Each process completely encapsulates its own state together with the code that operates on it and operates independently from other processes in the system, except for the data dependencies that are made explicit by channels. This allows for modular, scalable, and platform-independent application specifications.

System specification for KPNs is thus different from two other frequently used approaches for MPSoC software development, namely specification based on a board support package and specification based on a high-level application programming interface (API). When developing an application based on the board support package that is usually shipped with an MPSoC, the abstraction level is rather low. The focus is thus often on correctly implementing an application using low-level primitives for initialization, communication, or synchronization, rather than on optimizing an application. When using a high-level API the designer is relieved from dealing with low-level details (provided that the API has been ported to the target MPSoC). Compared to the KPN based approach, however, automatically optimizing programs written using a high-level API, such as MPI or OpenMP, is more difficult: Due to the lack of an underlying model of computation, the basis for automatically analyzing and optimizing a program is essentially missing.

3.2 System Synthesis

The Y-chart approach opens a gap between the system-level specification and the actual implementation of the design, sometimes referred to as the implementation gap. The challenge in bridging this gap is to preserve the KPN semantics on the one hand and achieve the desired performance on the other hand. Also, the pitfalls of parallel programming, such as deadlocks, starvation, and data races need to be handled. This is the task of system synthesis.

Different approaches for the synthesis of KPNs for software, hardware, and in combined hardware/software platforms have been proposed. The target architectures have been comprised of single-processors, multi-processors, and FPGAs. In all cases, system synthesis deals with the implementation of processes and channels as well as the arbitration of resources in case that processes and channels are mapped to shared resources. If not all parameters of an implementation are fixed before synthesis, the remaining degrees of freedom need to be exploited during system synthesis. In that case, system synthesis is often considered as an optimization problem where frequently considered optimization goals are the minimization of code size, the minimization of buffer requirements, or finding the schedules that minimize delays and maximize system throughput.

While many of these problems can be solved only for restricted subclasses, a few observations apply to general KPNs:

- First, KPN applications can be efficiently implemented on architectures with different processor, interconnect, and memory configurations, as shown in Table 2. As an example, KPN applications can be implemented on distributed memory (message-passing) architectures as well as shared memory architectures. The FIFO communication can be implemented using dedicated hardware FIFOs or buses, but also more complex communication topologies, such as hierarchical buses or networks-on-chip.

- Second, KPN applications can be executed in a purely data-driven manner based on their determinacy. This means that resources can operate independently from each other without any global synchronization. Pair-wise synchronization is only needed between processes that are directly connected by channels. From another perspective, this means that KPN applications can be scheduled with any scheduling policy that prevents deadlocks, i.e., preemptive, non-preemptive, or cooperative scheduling could be used. Due to these very relaxed requirements, KPN applications can usually be implemented easily on top of existing (real-time) operating systems. On the other hand, implementing a runtime-environment for a new platform from scratch is also possible because not many services need to be provided by the runtime-environment.
- Third, KPN applications can be easily partitioned into processes running in hardware and processes running in software. This is due to the parallel specification of the KPN application on the one hand, and due to the simple interaction of processes over FIFO channels on the other hand which facilitates the synthesis of the HW/SW interface.

The observations above indicate that synthesizing a KPN is conceptually not a very difficult task. Implementing a KPN based on a multi-processor operating system, for instance, is rather simple: Processes can be implemented as operating system processes or threads, and channels can be implemented using existing inter-process communication schemes. The difficulties in KPN synthesis origin from optimizing an implementation by minimizing the overhead for FIFO communication and the runtime environment. This can be achieved by considering low-level details of an implementation, for instance by efficiently using the hardware communication infrastructure (e.g. DMA engines) or by efficiently using the memory hierarchy (e.g. caches or scratchpad memories). On the other hand, optimizations can also be done at a high level, for instance, by (automatically) adjusting the granularity and topology of a KPN to the target architecture. This includes the replication of processes to increase the parallelism in a KPN or the merging of processes to reduce inter-process communication. We refrain from giving further details here and refer to the previous chapters for details on applicable techniques.

Finally, a further problem needs to be considered in the synthesis of KPNs: The denotational semantics of the Kahn model is based on FIFO channels with unbounded capacity. Since unbounded channels cannot be realized in physical implementations, however, KPNs need to be transformed in a way that allows for an implementation on channels with finite capacity. It can be shown that an operational semantics of KPNs based on channels with finite capacity matches the denotational semantics when artificial deadlocks can be avoided. An artificial deadlock is a deadlock caused by one or more channels having insufficient capacity. Due to the Turing-completeness of KPNs, it is in general not possible to determine sufficient channel sizes at design time, however. One possibility to deal with this situation are runtime approaches that detect and resolve artificial deadlocks during execution [25]. Another possibility is to restrict the communication behavior of the processes such that the channels become amenable to analysis at design time.

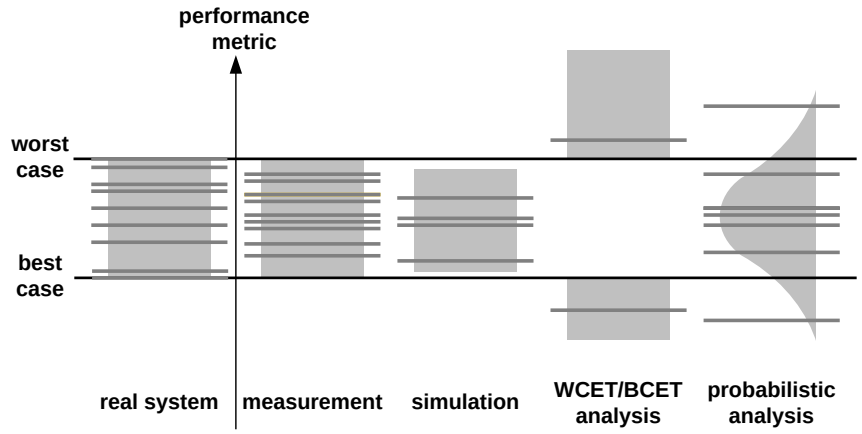


Fig. 2 Scope of different performance analysis methods for MPSoC.

3.3 Performance Analysis

During the design process, a designer is typically faced with questions such as whether the timing properties of a certain design will meet the design requirements, what architectural element will act as a bottleneck, or what the memory requirements will be. Consequently, one of the major challenges in the design process is to quantitatively analyze specific characteristics of a system, such as end-to-end delays, buffer requirements, throughput, energy consumption, or temperature rises due to application activities. We refer to this analysis as performance analysis.

The performance analysis of KPNs executing on MPSoCs poses a major challenge due to multiple and heterogeneous hardware resources, the distributed execution of the application, and the interaction of computation and communication on shared resources. To deal with these challenges, multiple methods have been successfully used in the context of KPN design flows. These methods differ in accuracy, evaluation time, set-up effort, and scope.

In Fig. 2, the scope of different performance analysis methods is compared. Leftmost, the interval of values for a performance metric as occurring in the real system is shown. This performance metric could be the end-to-end delay of a system, the utilization of a computation or communication resource, or the occupation of a channel buffer, for instance. Different performance analysis methods now differ regarding the values that can be obtained.

When taking measurements of the real system, the measured values only represent a subset of all possible values. Most likely, due to insufficient coverage of corner cases and the limited number of measurement samples, the interval bounds can only be estimated based on the measurements. This observation applies to simulation as well. Best-case and worst-case analysis methods take a different approach by providing safe results about the interval bounds, i.e. upper and lower bounds on

the worst-case and best-case behavior, respectively. On the other hand, usually not all parts of a system can be accurately modeled. In that case, (safe) optimistic and pessimistic assumptions need to be made, leading to bounds on system performance measures that are not tight. Finally, also probabilistic methods are used to provide quantitative statements about system behavior. In the following, we take a closer look at simulation and best-case/worst-case analysis due to their frequent application in KPN design flows.

Simulation is presumably the most frequently used method for performance analysis. This is reflected by the availability of a wide range of simulation tools that are applicable to different levels of abstraction. The most accurate but also slowest class are cycle-accurate simulators. Instruction-accurate simulators (also referred to as instruction-set simulators or virtual platforms) provide a good trade-off between speed and accuracy which allows entire MPSoCs to be modeled and simulated. An example is the so-called full system simulator of the Cell Broadband Engine which also allows switching between different simulation modes with different accuracies [26]. Besides performance analysis, virtual platforms can also be used for software development and debugging. For this purpose, the full system simulator of the Cell Broadband Engine provides a fast, purely functional simulation mode in which timing is not considered.

At higher levels of abstraction, also other kinds of simulation are used for performance analysis. One example is trace-based simulation in the Artemis design flow [11] or in DOL [27], for instance. In trace-based simulation, first an untimed execution trace of the application is recorded that contains computation and communication events of processes and channels. Based on an architecture description, the mapping of the application onto the architecture, and estimates about the time to process events, this trace is refined towards timing behavior. This technique allows designers to estimate the system performance. Depending on the level of detail in the trace and the modeling of the execution platform, estimation errors of less than 5 % have been reported with a significantly reduced simulation time compared to instruction-accurate simulation.

For the design of hard real-time systems, worst-case guarantees on the system timing need to be given. As stated above, worst-case bounds are difficult to obtain from simulation due to insufficient corner case coverage and often prohibitively long execution times of a simulation run. Therefore, analytic methods appear to be a promising method for providing worst-case guarantees even in the case of complex and large-scale MPSoC implementations. Prominent methods for analytic performance analysis are listed in the following.

- *Holistic Methods:* Holistic analysis is a collection of techniques for the analysis of distributed systems. The principle is to extend concepts of classical single-processor scheduling theory to distributed systems, integrating the analysis of computation and communication resource scheduling. Several holistic analysis techniques have been aggregated in the modeling and analysis suite for real-time applications (MAST) [28].
- *Compositional Performance Analysis Methods:* The basic idea of compositional performance analysis methods is to construct an analysis model of small com-

ponents and propagate timing information between these components. Typical components model the execution of processes on a processor, the transmission of data packets on interconnects, or traffic shapers. Timing information is described by event models, such as periodic, periodic with jitter and bursts, or more general models in terms of arrival curves. Prominent methods of compositional performance analysis are modular performance analysis (MPA) [29] and symbolic timing analysis for systems (SymTA/S) [30]. Both methods support a rich set of scheduling policies, such as preemptive and non-preemptive fixed priority scheduling, earliest deadline first scheduling, or time division multiple access. MPA is used in the DOL design flow, for instance.

- *Automata Based Analysis:* Performance analysis of MPSoCs has also been tackled using state-based formalisms. One example are timed automata [31]: The approach is to model a system as a network of interacting timed automata and formally verify its behavior by means of reachability analysis using the Uppaal model checker [32].

A comparison of these performance analysis methods is provided in [33]. Note that beside being suited for the analysis of real-time systems, analytic models are often used as the basis for performing system optimization, such as scheduling parameter optimization [34] or robustness optimization [35].

Finally, one can observe that none of the methods shown in Fig. 2 can fulfill all the requirements concerning accuracy, scope, and set-up effort. Therefore, combinations of the different methods have been proposed: Simulation has been coupled with native execution on the target platform to reduce simulation time [36] [37]. Different analytic methods have been coupled to broaden the analysis scope [38, 39]. Subsystems in simulation have been replaced by analytic models to reduce simulation time and eliminate the need to generate a detailed simulation model of a component [40]. In these efforts, the modularity of KPNs is often leveraged by using the FIFO channels as the interface between the different performance analysis methods.

3.4 Design Space Exploration

Designers of MPSoCs face a large design space due to the combinatorial explosion related to the available degrees of freedom. At several points in the design flow and at various levels of abstraction, they need to decide between design alternatives. Specifically, the design space of MPSoCs can be roughly divided into three domains: the application design space, the architecture design space, and the mapping design space. These three domains can be further split up, as shown in Fig. 3.

Exploration of the *application design space* can be split up into two main kinds of transformations, namely algorithmic and source transformations. Algorithmic transformations make explicit the coarse-grained parallelism in a sequential application by transforming it into a KPN. Given a KPN application, source transformations split and merge processes to trade-off parallelism and communication overhead.

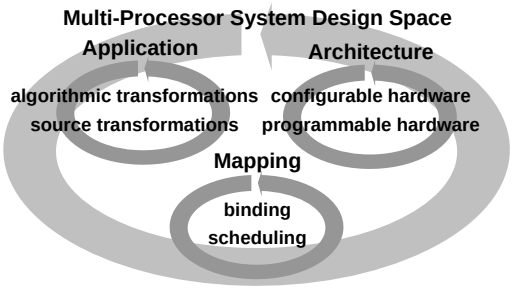


Fig. 3 Application, architecture, and mapping design space.

Exploration of the *architecture design space* attempts to find an optimized architecture for a given application. The goal is to instantiate programmable and (re-)configurable hardware components that allow an efficient implementation of an application.

Exploration of the *mapping design space* is the last step in the design space exploration. Given a KPN application and an architecture, processes and channels of the KPN are bound to processors and interconnects in the architecture, and scheduling policies are defined on shared communication or computation resources.

Usually, design space exploration is a multi-objective optimization problem. The goal is thus to find a set of Pareto-optimal designs which represent solutions with different trade-offs between the optimization goals such as performance, cost, energy consumption, or peak temperature. The final choice is left to the designer who needs to decide which of the Pareto-optimal designs to implement or to refine to the next level of abstraction.

Available approaches to the exploration of design spaces can be characterized as follows. Gries [41] presents a more detailed survey of automated design space exploration and performance analysis in different design flows.

- *Manual Exploration:* The selection of design points is done by the designer. When taking this approach, the advantage of using a KPN design flow lies in efficient performance analysis and automated synthesis of selected designs.
- *Exhaustive Search:* All design points in a specified region of the design parameters are evaluated. Very often, this approach is combined with local optimization in one or several design parameters in order to reduce the size of the design space. Due to the availability of fast performance analysis techniques for KPNs, exhaustive search is a realistic option if the design space is limited (or can be pruned) to roughly a few thousand designs.
- *Reduction to Single Objective:* For design space exploration with multiple conflicting criteria, there are several approaches available that reduce the problem to a single criterion optimization. For example, manual or exhaustive sampling is done in one (or several) directions of the search space and a constraint optimization, e.g. iterative improvement or analytic methods is done in the other. One may also combine the various objectives to a single criterion by means of a weighted sum where the weights express the preferences of the designer.

- *Black-box Randomized Search:* The design space is sampled and searched via a black-box optimization approach, i.e. new design points are generated based on the information gathered so far and by defining an appropriate neighborhood function (variation operator). The properties of these new design points are estimated which increases the available information about the design space. Examples of sampling and search strategies are Pareto simulated annealing, Pareto tabu search, or evolutionary multi-objective optimization. These black box optimization methods are often combined with local search methods that optimize certain design parameters or structures. This approach is most frequently used in KPN design flows, as illustrated in Table 2.
- *Problem-Dependent Approaches:* In addition to the above methods, one can find also a close integration of the exploration with a problem-dependent performance analysis of implementations. This approach is often used in design flows that are based on subclasses of KPNs. The StreamIt and MAMPS design flow, for instance, are based on SDF (Synchronous Data Flow) graphs and use adopted techniques for design space exploration, see chapter [42].

4 Specification, Synthesis, Analysis, and Optimization in DOL

Until now, this chapter introduced KPN design flows and the corresponding main design activities. This section will provide additional technical details by means of a concrete example of a typical design flow: the Distributed Operation Layer (DOL) [12] [43]. The underlying concepts for system specification, synthesis, performance analysis, and design space exploration will be considered, as well as a few typical experimental results for the size of the implementation, the runtime, and accuracy of the applied methods. DOL is currently being extended towards scenario-based design flow [44], and supports the design, optimization, and simultaneous execution of multiple dynamic applications on a MPSoC starting with a similar programming model as [45]. However, this section does not discuss these extensions, focusing on the typical design flow for single Kahn process networks.

4.1 Distributed Operation Layer

The distributed operation layer (DOL) [12] [43] is a platform independent MPSoC design flow based on the Kahn process network (KPN) model of computation [3] and targeted at real-time multimedia and (array) signal processing applications.

The DOL design cycle, as shown in Fig. 4, follows the Y-chart approach in which the application specification is platform-independent and needs to be related to a concrete architecture by means of an explicit mapping. As usual, the design starts with the specification of the application and architecture (and sometimes even a mapping). Then, code for the functional simulation of the application is automati-

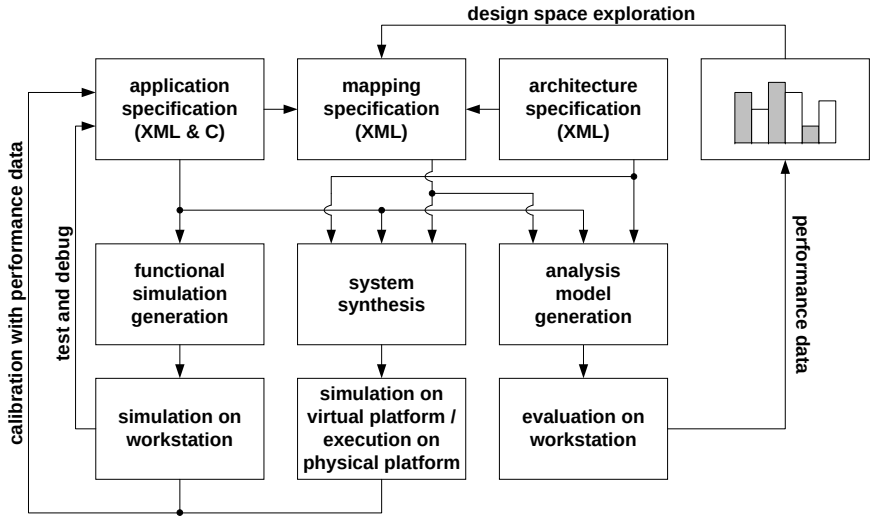


Fig. 4 Overview of the DOL design flow.

cally generated for testing and debugging the parallel application code with standard debugging tools on a standard PC/workstation.

Once the application is functionally correct, it can be mapped onto the target architecture. Based on the architecture and the mapping specification, the system is synthesized by generating the corresponding binaries. Note that, here, system synthesis refers to software synthesis only as the architecture specification is considered to be unaltered during the exploration phase. Then, the synthesis involves the generation of the mapping-dependent source code for processors, the compilation, and the linking to platform specific libraries as well as to the run-time environment. Generated binaries can either be executed on a simulator of the target platform or on the real MPSoC. Both, the functional and the low-level simulation provide performance figures that will enrich the application specification. This information will be used in later phases for the calibration of the analysis model.

The design flow described so far is typical for MPSoC design and very similar to the other design flows listed in Table 1 and explained in the previous chapters. What is different in DOL is its focus on the design and analysis of real-time signal processing applications. To this end, an analytic worst-/best-case performance analysis method has been embedded into the design flow. Besides enabling the analysis of real-time systems, using an analytic method for performance analysis facilitates rapid design space exploration due to short analysis times. The resulting performance data are embedded in a design space exploration loop in search of the optimal mapping.

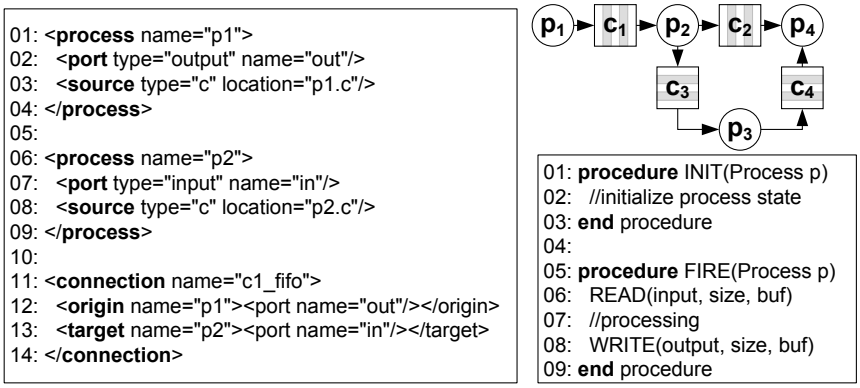


Fig. 5 Kahn process network model. Left: XML description of the process network structure. Right top: example of a process network. Right bottom: C code of individual processes.

4.2 System Specification

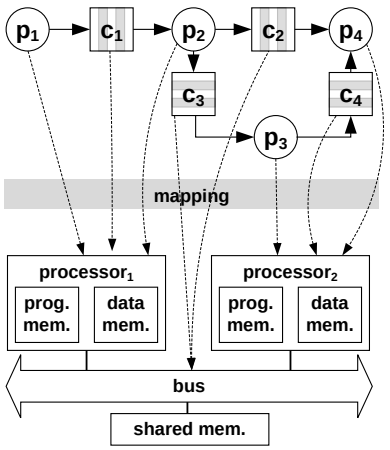
For designing the specification format of an MPSoC, one has to consider three criteria. First, the specification format should be expressive enough to represent the class of envisioned applications, i.e. (real-time) signal processing applications. Second, the specification should facilitate automation of system synthesis and analysis. The third criterion is the possibility of mapping an application in different ways onto an architecture. In the DOL framework, these criteria are met by specifying the application as a Kahn Process Network [3] and by specifying the application independently of architecture.

When designing parallel applications irrespective of architectures, an important feature is the ability to specify different topologies of the process network with different degrees of parallelism. For this reason, the KPN coordination part is kept separately (described in XML) from the source code of the individual processes (described in C/C++), see Fig. 5. Similar hybrid XML/C formats are employed by other frameworks as well (e.g. in Artemis [11], ESPAM [13], and MAMPS [15]).

While the syntax of the XML file is specified using an XML schema, the C code is based on a simple API. As shown in Fig. 5, this API basically consists of four functions, two of which concern computation, namely `INIT` and `FIRE`, and two of which concern communication inside the `FIRE` procedure, namely `READ` and `WRITE`:

- `INIT` contains the code that is executed once at start-up to initialize a process.
- `FIRE` contains the code that is repeatedly called by the scheduler.
- `READ` implements the blocking read from a FIFO channel.
- `WRITE` implements the blocking write to a FIFO channel.

A similar API is defined by Y-API [46], for instance, a library for specifying and executing Kahn process networks.




```
01: <binding name="b_p1" type="computation">
02:   <process name="p1"/>
03:   <processor name="processor1"/>
04: </binding>
 ...
16: <binding name="b_c2" type="communication">
17:   <sw_channel name="c2"/>
18:   <writepath name="dmem1_bus_shm"/>
19:   <readpath name="shm_bus_dmem2"/>
20: </binding>
...
30: <schedule name="sched1" type="fixedpriority">
31:   <resource name="processor1"/>
32:   <origin name="p1">
33:     <configuration name="priority" value="0"/>
34:   </origin>
35:   <origin name="p2">
36:     <configuration name="priority" value="1"/>
37:   </origin>
38: </schedule>
```

Fig. 6 Mapping of a Kahn process network onto a two-processors architecture and an example of a corresponding mapping XML file.

The architecture model in DOL is an abstract representation of the underlying execution platform. Its purpose is to determine at a system-level the consequences of the application mapping. This abstract architecture models the topology (i.e. the set of processors and communication paths between processors) and includes performance figures of the underlying platform useful for performance analysis, e.g. the clock frequency and throughput of architectural resources. The architecture model is a structural description that does not express the functional behavior, and which is specified in XML, similar to the application model. This XML architecture representation is not specific to DOL, but also encountered in other frameworks, such as Artemis and MAMPS.

The application model is brought in correspondence to the architecture model by a mapping (see Fig. 6) which can be either established manually by an experienced designer or generated automatically by design space exploration. This mapping fixes the allocation of hardware resources, the binding of the application elements onto these resources, and the scheduling on shared resources. For the mapping specification, once more, the XML format is used. The mapping XML serves as intermediate format and interface between tools, i.e. the design space exploration tool generates a mapping XML as an output, which is the input for the software synthesis tool.

The application XML, the architecture XML, and the mapping XML are the basis for the following DOL synthesis steps, i.e. for the functional simulation and the implementation of the final MPSoC, but also for the generation of the analytic performance analysis model (see Fig. 4).

4.3 System Synthesis

Similar to other frameworks, an application specified in DOL cannot be directly executed by just compiling the provided source code of the processes. A synthesis step is required that generates the "glue code" implementing the processes and channels, the bootstrapping and the scheduling of the application. Specifically, synthesis is done first for a standard PC/workstation to support the functional verification and debugging of the application (in which case it should be rather termed functional simulation generation) and second for the target MPSoC (which is properly known as system synthesis). However, due to similarities, the two steps are treated together in the following subsections as facets of system synthesis in the DOL design flow. Note that when the behavior of a part of an application can be restricted to a subclass of KPN, the general approach described below could be combined with one of the corresponding synthesis techniques described in the previous chapters.

4.3.1 Functional Simulation Generation

The purpose of providing a functional simulation that can be executed on a standard PC/workstation is to provide the application developer with a convenient approach to test and debug the application. Specifically, functional bugs within the application can be exposed and debugged by running a functional simulation on a standard PC and using standard debugging tools, e.g. the GNU debugger gdb.

A second role of the functional simulation is to obtain architecture-independent application parameters for performance analysis, such as the amount of data transferred between processes or the number of activations of processes. These parameters can easily be obtained from functional simulation by monitoring the calls of the `READ`, `WRITE`, and `FIRE` methods. By back-annotating these parameters to the application specification as shown in Fig. 4, they can be referred to during performance analysis, as explained later in this section.

Using the DOL design flow, a functional simulation can be automatically generated according to the application specification: For each process, an execution thread is instantiated. To implement software channels, inter-thread communication channels are used. The execution of the application is then controlled by a simple data-driven scheduler. Since Kahn process networks are determinate, this is a viable possibility because the scheduling does not influence the input/output behavior of the application.

In DOL, the functional simulation is based on the SystemC library. Therefore, processes can be implemented as user-space threads which incurs less runtime overhead compared to using an operating system thread library, such as the pthread library. Fig. 7 shows the software architecture of the functional simulation based on SystemC: Each Kahn process is embedded into a SystemC thread, whereas each Kahn software channel is implemented as a SystemC channel. Moreover, the main file that bootstraps the process network and implements the scheduler to coordinate the quasi-parallel execution of processes is generated automatically as well.

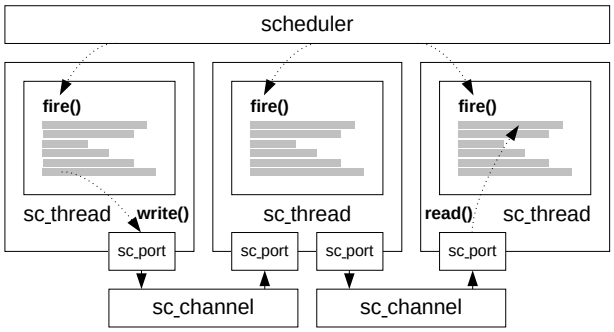


Fig. 7 Software architecture of the functional simulation of a KPN application based on SystemC.

Another frequently chosen library is the pthreads library. On multicore multiprocessors where single operating system threads can be executed on different cores, a functional simulation based on pthreads can even achieve a speed-up compared to the sequential version of the application. In [47], speeds-ups of more than 3 have been reported for executing applications specified in SHIM on a quad-core Intel Xeon processor.

4.3.2 Software Synthesis

After the application has been functionally verified by functional simulation, it is ported to the target platform. This requires an architecture dependent runtime environment in which the application is executed. The role of the runtime environment is to hide architectural details of the MPSoC platform by providing a set of high-level services enabling the execution of an application on the platform, such as task scheduling, inter-process communication, or inter-processor communication. Depending on the target platform, developing (parts of) the runtime environment might be necessary to create the basis for software synthesis.

In case of the DOL design flow, different hardware MPSoC platforms are supported:

- *Cell Broadband Engine [48]:* MPSoC consisting of a PowerPC-based Power Processor Element and eight DSP-like Synergistic Processing Elements interconnected via a ring bus.
- A*tmel Diopsis 940 [49]:* tile-based MPSoC, where a single tile is composed of an ARM9 processor and a DSP interconnected by an AMBA bus; up to eight tiles are interconnected via a network-on-chip.
- *MPARM [50]:* Homogeneous MPSoC consisting of identical ARM7 processors connected by an AMBA bus.
- *Intel SCC [51]:* Many-core homogeneous architecture with 48 cores organized in 24 tiles, each tile embedding two cores. Tiles are connected via a mesh on-chip network, and each tile has also a message passing buffer.

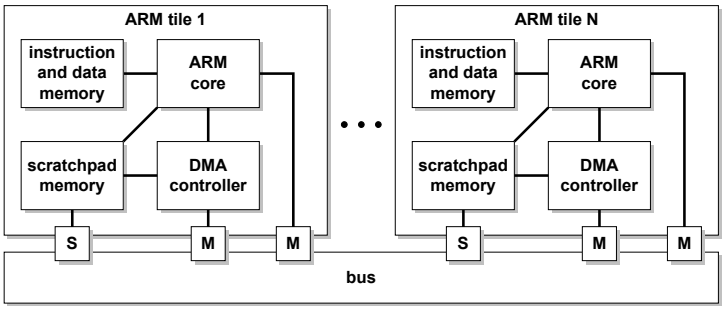


Fig. 8 Block diagram of the MPARM architecture.

Fig. 8 depicts a block diagram of the MPARM architecture. Software synthesis for MPARM is based on the RTEMS (Real-Time Executive for Multi-processor Systems) [52] operating system. Basic services provided by RTEMS are the scheduling of processes, device drivers for inter-process communication, and device drivers for system input/output. Based on these services, it is rather simple to bootstrap and execute a process network. As an example, Listing 1 illustrates parts of the code for bootstrapping a process network based on the RTEMS API. Software synthesis for the Cell Broadband Engine is described in [53] in the context of the DOL design flow, and in [23] in the context of the SHIM design flow, for instance.

Listing 1 shows parts of a main file for a producer-consumer type application running on MPARM. In lines 2-3, memory is allocated for the local data of the producer and consumer processes. In lines 6-9, two tasks are created for the processes by allocating a task control block, by assigning a task name and a task ID, by allocating a stack, and by setting initial attributes like the task priority and the task mode. Lines 11–12 show the creation of a message queue. In lines 14–18, the `rtems_task_start` directive puts the tasks into the ready state, enabling the scheduler to execute them. Finally, the initialization tasks deletes itself (line 20).

Listing 1 RTEMS initialization task in which two tasks are bootstrapped to run a producer and consumer process of a process network.

```
1  rtems_task Init(rtems_task_argument arg) {
2    producer_wrapper ← malloc(sizeof(RtemsProcessWrapper));
3    consumer_wrapper ← malloc(sizeof(RtemsProcessWrapper));
4
5    for (j ← 0; j < 2; j++) {
6      status ← rtems_task_create(j + 1, 128,
7                 RTEMS_MINIMUM_STACK_SIZE, RTEMS_DEFAULT_MODES,
8                 RTEMS_DEFAULT_ATTRIBUTES, &(task_id[j]));
9    }
10
11   status ← rtems_message_queue_create(1, 10, 1,
12              RTEMS_DEFAULT_ATTRIBUTES, &queue_id[0]);
13
14   status ← rtems_task_start(task_id[1], producer_task,
```

```
15              (rtems_task_argument)producer_wrapper);
16
17      status ← rtems_task_start(task_id[2], consumer_task,
18              (rtems_task_argument)consumer_wrapper);
19
20      rtems_task_delete(RTEMS_SELF);
21  }
```

For all the mentioned platforms, the main challenge is to provide an efficient FIFO channel implementation that allows overlapping computation and communication in order to reduce the runtime overhead as much as possible. Aspects that play an important role in this context are the size and location of channel buffers, the efficient use of DMA controllers for data transfers between processors, and the minimization of synchronization messages.

4.4 Performance Analysis

The DOL design flow is targeted towards the design of real-time multi-media and signal processing applications. These systems must meet real-time constraints, which means that not only the correctness and performance of a system are of major concern but also the timeliness of the computed results. Typical questions in this context are:

- What is the response time to certain events? Is this response time within the required real-time limits?
- Can the system accept additional load and still meet the quality-of-service and real-time constraints?
- Is a system schedulable, that is, are all real-time constraints met?

To be able to answer these questions, a suitable combination of system design and performance analysis is required. To this end, it is essential that the architecture, application, and runtime-environment of a system are amenable to formal analysis, because simulation or measurements are not able to provide guarantees about timing properties. On the other hand, performance analysis methods with a reasonable scope and accuracy need to be employed such that effects occurring in the system implementation can be faithfully modeled. For MPSoC applications, this includes the modeling of heterogeneous resources and their sharing, the modeling of complex timing behavior arising from variable execution demands and interference on shared resources, or the modeling of different processing semantics.

Many approaches have been proposed to solve this problem, see [54] for an overview. Frequently used approaches are time-triggered and synchronous approaches, for instance. In purely time-triggered approaches, such as the time-triggered architecture [55] or Giotto [56], processing time of resources is allocated to tasks in fixed time slots. This fixed allocation facilitates analysis, but dimensioning of the slots turns out to be difficult for varying workloads. For instance, using the worst-case

workload for setting the slot sizes might lead to over-dimensioned systems. Purely synchronous approaches implemented in synchronous languages, such as Esterel, Lustre, and Signal, rely on a global clock that divides the execution of a system into a sequence of atomic processing steps [57]. While synchronous approaches are successfully used for single-processor systems, applying them to MPSoCs is difficult because MPSoCs are usually split up into different (asynchronous) clock domains such that the synchronous assumption does not hold.

The approach taken in DOL relies on using compositional performance analysis where a system is modeled as a set of processing components that interact via event streams. This is a good match for MPSoCs as well as for KPNs. Contrary to other approaches, the approach is rather flexible in that it is not limited to a certain system architecture, scheduling policy, or execution semantics. Specifically, Modular Performance Analysis (MPA) is integrated into the DOL design flow. In the following, the basic concepts of MPA are reviewed. Afterwards, it is summarized how MPA is integrated into the DOL design flow.

4.4.1 Modular Performance Analysis (MPA)

MPA [29, 58] is an analytical approach for the analysis of real-time systems. It is based on real-time calculus [59] which has its foundations in network calculus, a method for worst-case analysis of communication networks [60, 61]. With MPA, hard upper and lower bound for performance metrics of a distributed real-time system can be computed. As shown in Fig. 9, the performance model of a system is decomposed into a network of abstract processing components that model the computation and communication in a system. These processing components are connected by abstract event streams that model the timing properties of the data streams flowing through the system. Finally, resources are modeled by resource streams that model the availability of processing resources to computation and communication tasks. Different scheduling policies can be modeled by differently connecting processing components and resource streams. As an example, Fig. 9 illustrates fixed priority (FP) scheduling on processors and time-division multiple-access (TDMA) scheduling on the bus.

The processing components modeling computation and communication are characterized by the worst-case/best-case execution demand and the minimal and maximal token size, respectively. Event streams are characterized by so-called arrival curves and resource streams by service curves. Summarizing, these abstractions allow the modeling of computation and communication on heterogeneous resources in a unified manner.

Based on these abstractions, the system is analyzed by consecutive propagation of event streams between components. Depending on the mapping of processing components to a resource and its scheduling/arbitration, the timing properties of the streams change. System properties such as resource utilization, system throughput, end-to-end delays, or buffer sizes can be derived this way. Tool support for actually

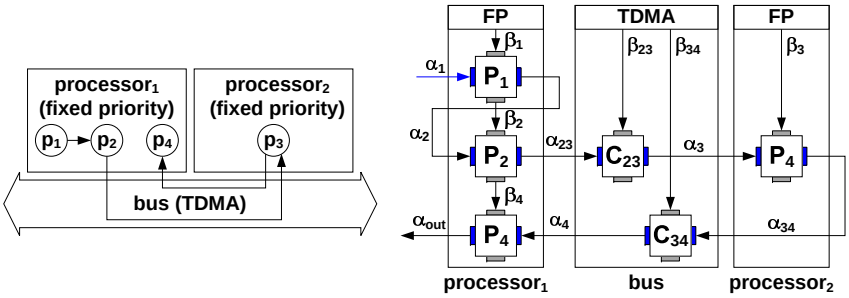


Fig. 9 MPA model of a system with two processors connected by a bus on which a KPN with four processes is executing. In the MPA model, horizontal edges represent event streams whereas vertical edges represent resource streams.

performing the analysis is provided by a freely available Matlab toolbox [62] that implements the underlying algebraic operations.

4.4.2 Integration of MPA into the DOL Design Flow

It has been mentioned that the goal of system synthesis is to bridge the implementation gap, that is, to refine a high-level system specification into an actual system implementation. Similarly, there is an "abstraction gap" between an MPA model and the implementation: The execution of sequential processes on a processor is modeled by an abstract processing component, the availability of resources is abstracted by service curves, and the dataflow through the systems by arrival curves. Bridging this abstraction gap, that is, creating an analysis model that correctly models the implementation is a non-trivial task. On the one hand, the high-level system specification is conceptually similar to the analysis model but does not contain all the parameters required to generate an MPA model, such as best-case/worst-case task execution times or token sizes. The implementation, on the other hand, implicitly contains this information, but extracting the information is not straightforward.

In the DOL design flow, the abstraction gap is bridged by analysis model generation and calibration. In model generation, the high-level system specification is translated into a corresponding MPA model. In model calibration, the required model parameters are obtained. In both steps, the modular structure of the application and architecture specification is leveraged. The basic approach is depicted in Fig. 10. The analysis model generation represents a branch in the design flow that is parallel to the system synthesis. The analysis model calibration makes use of this feature later on in order to build a database with necessary performance data for the formal model. In the following, the basic approach is described. Further details are provided in [63].

The goal of model generation is to translate the high-level application, architecture, and mapping specification into a corresponding MPA model implemented as a

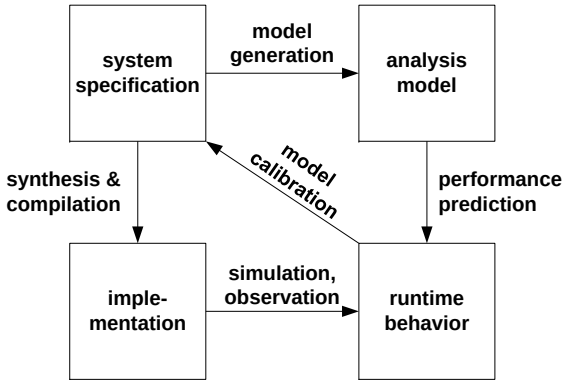


Fig. 10 Analysis model generation and calibration in the DOL flow.

Matlab script. Due to the modular specification of the application that is made explicit in the process network XML description, this is straightforward: Each process is simply modeled as an abstract processing component. Similarly, the communication channels between processes are modeled as abstract communication components and connected to the processing components according to the topology of the KPN.

The aim of model calibration is to obtain the quantities to parameterize the generated model such that it correctly models the implementation. Basically, three different types of parameters can be distinguished:

- First, there are the application parameters that are architecture and timing independent. An example is the minimal and maximal size of tokens transmitted over each channel.
- Second, there are the parameters that depend only on the architecture and the runtime environment. Examples are the throughput of the different communication resources or the context switch time of the runtime environment.
- Third, there are the application parameters that depend on the architecture and the mapping. Basically, whenever the architecture or mapping changes, new parameters need to be determined. An example is the worst-case/best-case execution time of a process on a processor.

Depending on the parameter type, there are different ways to obtain them. Timing independent parameters can be obtained from the functional simulation, due to the determinism of KPN applications. Architecture dependent parameters need to be obtained once a new hardware architecture or runtime environment is employed for realizing a system. The parameters of the third category, i.e. application parameters that depend on the architecture and the mapping, are more difficult to determine. Similar to system-level performance analysis, different methods for worst-case/best-case execution time analysis have been proposed, for instance [64]. In the DOL design flow, timed simulation on a virtual platform is employed. Note that compared to formal methods, this approach is not suitable for the calibration of hard real-

time system models unless complete coverage of corner cases is exhibited in the calibration simulation runs. One can observe that similar approaches are taken in other design flows. In the Artemis design flow, for instance, model generation and calibration is used to create a model for trace-based simulation [65].

Finally, the DOL framework have been extended with capabilities for worst-case thermal analysis, as nowadays providing guarantees on maximum temperature is as important as functional correctness and timeliness. Aware of the performance-temperature correlation, DOL is optimizing the system design with respect to both worst-case performance and worst-case temperature, analyzed in the same MPA framework. The basic worst-case peak temperature analysis method in MPA for a single processor under a broad range of uncertainties in terms of task execution times, task invocation periods, and jitter in task arrivals is described in [66]. Extensions are then proposed in [67] for analysis of MPSoC platforms by considering both the self-heating of the processor and the heat transfer between neighboring processors. In the same manner as it is done for timing, thermal analysis models are automatically generated from the same set of specifications as used for software synthesis. To increase the model accuracy, both analysis models are calibrated with data corresponding to real system parameters obtained in an automatic manner, prior to design space exploration. The calibration tool-chain for the thermal model is described in [68].

4.5 Design Space Exploration

The final piece of the DOL flow is design space exploration, built on top of analysis and synthesis tools to find an optimal mapping. In general, the problem of optimally mapping an application to a heterogeneous distributed architecture is known to be NP-complete. Even for systems of modest complexity, one thus needs to resort to heuristics to solve the problem. In addition, the mapping problem is usually multi-objective such that there is no single optimal solution but a set of Pareto-optimal solutions constituting a so-called Pareto-front.

In DOL, the aim of the design space exploration is to compute the set of Pareto-optimal solutions representing different trade-offs in the design space. Based on the (approximated) Pareto-front, the designer chooses the final solution to implement. Therefore, the mapping problem is specified as a multi-objective optimization problem.

Formally, a multi-objective optimization problem is defined on the decision space X which contains all possible design decisions, i.e. architectures, applications and mappings. To each implementation $x \in X$ there is associated an objective vector f in the objective space Z that consists of n objectives $f = (f_1, \ldots, f_n)$ which should be minimized (or maximized). An order relation $\leq$ is defined on the objective space Z, which induces a preference relation $\preceq$ on the decision space X: $x_1 \preceq x_2 \Leftrightarrow f(x_1) \leq f(x_2)$, for $x_1, x_2 \in X$. In other words, for the mapping problem

for instance, if mapping x_1 is better (minimal) in all objectives than mapping x_2, the optimization algorithm will "preferentially" select mapping x_1.

The design search space, symbolized with Ψ, is the set of all subsets of X, i.e. it includes all possible solution sets $A \subseteq X$. The final goal is to determine an optimal element of Ψ, i.e. an optimal subset of all possible implementations X. This subset should reflect all trade-offs induced by the multiple objectives. The preference relation $\preceq$ on X that has been defined above can now be used to define a corresponding set preference relation, symbolized with $\preccurlyeq$, on the search space Ψ.

This set preference relation provides the information on the basis of which two candidate Pareto sets can be compared: $A \preccurlyeq B \Leftrightarrow \forall b \in B, \exists a \in A : a \preceq b$. This property reflects the concept of Pareto-dominance: A design point dominates another one if it is equal or better in all criteria and strictly better in at least one. Moreover, the search in the design space will be pursued until a good Pareto-optimal set approximation $A \in \Psi$ is found.

In DOL, evolutionary algorithms are used to solve the mapping optimization problem. Evolutionary algorithms find solutions to a given problem by iterating two main steps [69]: (1) selection of promising candidates, based on an a-priori evaluation of candidate solutions and (2) generation of new candidates by variation of previously selected candidates. The principle of the selection in evolutionary multi-objective optimization is sketched in Algorithm 1. For a complete description, we refer to [70]. The starting point is a randomly generated population $P \in \Psi$ of size m. During optimization, a heuristic mutation operator based on selection and variation generates another set $P' \in \Psi$, which is wanted to be better than P in the context of the predefined set preference relation $\preccurlyeq$, i.e. $P' \preccurlyeq P$. Finally, P is replaced by P', if the later is preferable to the former (i.e., $P' \preccurlyeq P$), or P it remains unchanged in the opposite case.

Algorithm 1 Main optimization function.

```
 1: randomly choose A ∈ Ψ                          ▷ generate initial set P of size m
 2: set P ← A
 3:
 4: while termination criterion not fulfilled do    ▷ main optimization loop
 5:     P' ← heuristicSetMutation(P)
 6:     if P' ≼ P then
 7:         P ← P'
 8:     end if
 9: end while
10: return P
```

The heuristic set mutation operator is detailed in Algorithm 2. First, k new solutions are created based on P, after an appropriate selection and variation operation. While the variation is problem-specific, the selection is independent of the problem, using either an uniform random selection or a fitness-based selection. Then, the k new solutions are added to P, resulting in a set P' of size $m+k$. P' is iteratively truncated to size m by removing the solutions with worst fitness values. Note that

Algorithm 2 Heuristic set mutation function.

```
1: function HEURISTICSETMUTATION(()P)
2:     generate {r1,...,rk} ∈ X based on P
3:     P' ← P ∪ {r1,...,rk}
4:     while |P'| > m do
5:         choose p ∈ P' with fitness(p) = min_{a∈P}{fitness(a)}
6:         P' ← P' \ {p}
7:     end while
8:     return P'
9: end function
```

the fitness values are associated in a performance evaluation process, i.e., in DOL we use the MPA framework as described in Section 4.4.

While the selection algorithm described is domain independent, specific methods are used to include domain specific knowledge into the search process and select "best" solutions among a population. These are the domain representation (i.e., the system mapping), the evaluation of designs (i.e., the MPA analysis), and the variation of a population of solutions by mutation and cross-over operations.

Although standard variation schemes exist for mutation and crossover, their implementation is strongly dependent on system properties. The mutation generates a local neighborhood of selected design points. In the DOL context, the mutation affects the mapping solutions; for instance, different mappings can be generated with different bindings of processes onto processors. The crossover recombines two selected solutions to generate a new one. Note that during mutation or crossover, infeasible (mapping) solutions can be generated. In this situation, a repair strategy is invoked, which, in conformity with the evolutionary algorithm principle, attempts to maintain a high diversity in the population. An example could be the rerouting of inter-process communication, when during re-mapping a process was bound to a new location.

In DOL, the design space exploration framework includes several tools, as shown in Fig. 11. In particular, the EXPO [71] tool is the central module of the framework. As underlying multi-objective search algorithm, Strength Pareto Evolutionary Algorithm (SPEA2) [72] is used that communicates with EXPO via the PISA [73] interface. Similarly, the design space exploration framework in Artemis [11] is also based on PISA and SPEA2.

Using the frameworks of EXPO and PISA relieves the designer from implementing those parts of the design space exploration that are independent of the actual optimization problem. For example, the selection may be handled inside the multi-objective optimizer SPEA2. The designer just needs to focus on the problem specific parts, that is, the generation, variation, and evaluation of solutions. The implementation of problem specific parts starts with the specification, where the application and architecture (and later on, the mapping) are automatically extracted from the corresponding XML files and represented in the design space exploration framework. Then, candidate mappings are inspected (as described above) based on the provided variation methods. Finally, during design space exploration, the objective

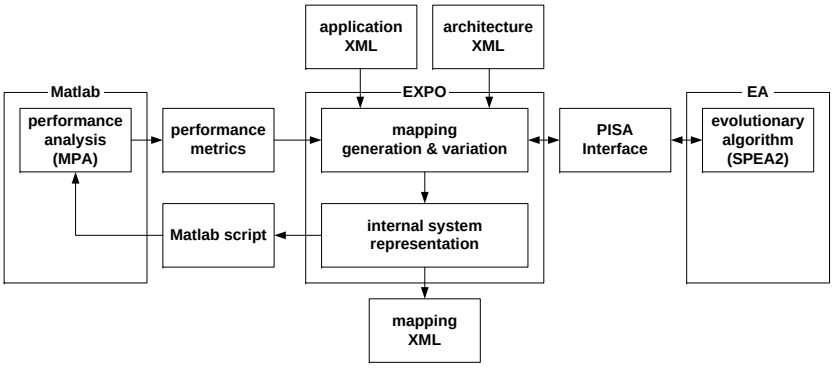


Fig. 11 Design space exploration in the DOL framework.

values of all candidate mappings are computed by generating the corresponding Matlab MPA scripts and interfacing Matlab for their evaluation. In DOL, all these operations are automatically parameterized using the application and architecture specification. Note that the approach described above is a heuristic search procedure. Therefore, it does not guarantee the optimality of the final solution, i.e., the final set of solutions. However, in our experiments we have identified that after several design space exploration cycles, the found solutions are close to optimal even for large problem complexities.

4.6 Results of the DOL Framework

In this section, a few results are highlighted that have been obtained by applying the DOL design flow described above. Specifically, the design and analysis of a Motion-JPEG (MJPEG) decoder [74] running on MPARM [50] is considered. For the execution of the system, we used a 31-frame input bitstream encoded using the QVGA (320×240) YUV 444 format.

The MJPEG decoder decompresses a sequence of frames by applying JPEG decompression to each frame. Because of the inherent parallelism in the MJPEG algorithm, the decoding is done in a pipeline with five stages, each stage being implemented as a Kahn process. The first and last stages are the splitting of streams into frames (`ss`) and the merging of frames back to streams (`ms`). The variable length decoding and the splitting of frames into macroblocks form the second stage (`sf`). The zigzag scan, inverse quantization and the inverse discrete cosine transform form the third (`zii`), while combining macroblocks back to frames forms the fourth stage (`mf`).

Using the design space exploration framework of DOL based on the PISA interface and SPEA2, one can compute the Pareto optimal mappings of the MJPEG application onto an MPARM system with a variable number of processors. The

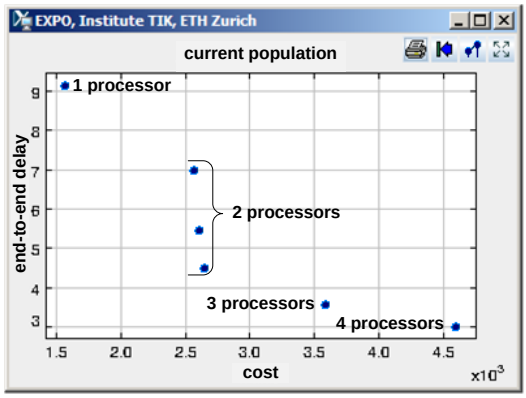


Fig. 12 Pareto optimal solutions resulted after the design space exploration (screenshot of the EXPO tool).

mapping has been optimized in conformity with two design objectives: (1) end-to-end delay in the system computed as the result of MPA analysis, which is an upper bound of the actual end-to-end delay and (2) the cost of the system evaluated as a sum of costs associated with the used processors, memories, and the bus. In the experiments, a population size of 60 individuals has been chosen and the algorithm has been executed for 50 generations. These parameters generally depend on the complexity of the problem to solve. The obtained Pareto front is shown in Fig. 12, consisting of 6 mapping solutions onto a different number of processors. The search in this design space took about 2 hours.

For illustration purposes only, we employ a simple configuration with a small number of processes that can be mapped in different ways onto the architecture and can communicate via different hardware communication paths. However, a more efficient implementation can be obtained if the same application is specified with a scalable number of processes processing data in parallel. This would enlarge the parallelism of the design but also the dimensionality of the design space. A more complex design space exploration with the DOL framework is shown in [12], where a scaled version of an MPEG-2 decoder has been mapped onto a tile-based heterogeneous architecture.

Other KPN flows, like Sesame in the Artemis project [11] that use exactly the same optimization frameworks of PISA and SPEA2, report comparable parameters and results for the design space exploration. However, their design space exploration is considerably shorter, i.e. 5 s for a design with 8 processes, because they use a much simpler additive performance model. Of course, for larger problem sizes all the parameters will scale and the design space exploration can take much longer if more accurate analysis methods, like MPA or simulation, are used. However, this is an acceptable cost since designers are exploring the entire design space only once.

In the remainder of this section, a mapping of the MJPEG application onto a 3-tile MPARM system, that is, three ARM processors interconnected via a shared AMBA bus, is considered. The resulting MPA model is shown in Fig. 13.

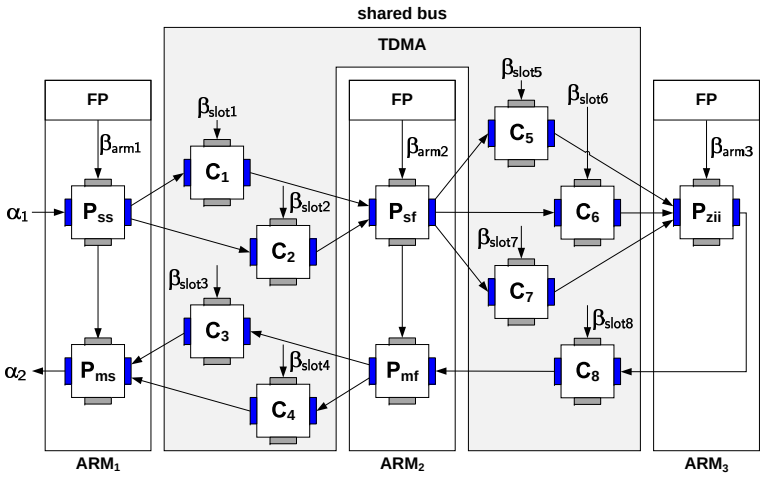


Fig. 13 MPA model of MJPEG application mapped onto a 3-tile MPARM platform. P_x are the five processes of the MJPEG decoding pipeline that communicate via the C_y software channels.

To evaluate the efficiency of the design flow (i.e., the time spent in obtaining results), Table 3 lists the durations of the different design steps for performance analysis of the different design solutions. Several conclusions can be drawn:

- Automated software synthesis can be done fast. Actually, most of the time required to generate the functional simulation or the binaries for the target platform is required to compile the generated source code rather than to generate this code.
- The table shows that timed simulation on the virtual platform is the most time-consuming step in the design flow. Minimization of simulation time is thus paramount and actually possible in a systematic design flow, as has been shown above. Conversely, simulation time can become a major bottleneck in MPSoC design when following a less systematic design flow requiring many design iterations involving timed simulation.
- The generation and analysis of a system's MPA model is a matter of seconds. Note that similar times have been reported for alternative performance analysis methods like trace-based simulation in the Artemis design flow, for instance. While further reducing this time is desirable, it is a reasonable time frame for performance analysis within a design space exploration loop.
- The one-time calibration to obtain the parameters for the MPA model takes several seconds albeit being completely automated. Extracting these parameters manually would be a major effort.

In order to evaluate the accuracy of MPA estimations, the performance bounds computed with MPA are compared to actual (average-case behavior) quantities observed during system simulation. The differences are in a range of 10-20%, which is typical for a compositional performance analysis. Differences in the same range have been observed for several systems in [33], for instance. There are two main rea-

Table 3 Duration of different design steps in the MJPEG design, measured on a 1.86 GHz Intel Pentium Mobile machine with 1 GB of RAM. The simulations were executed to decode a 31-frame input bitstream encoded using the QVGA (320×240) YUV 444 format.

step		**duration**
model calibration (one-time effort)	functional simulation generation	42 s
	functional simulation	3.6 s
	synthesis (generation of binary)	4 s
	simulation on MPARM	13550 s
	log-file analysis and back-annotation	12 s
model generation		1 s
performance analysis based on generated model		2.5 s

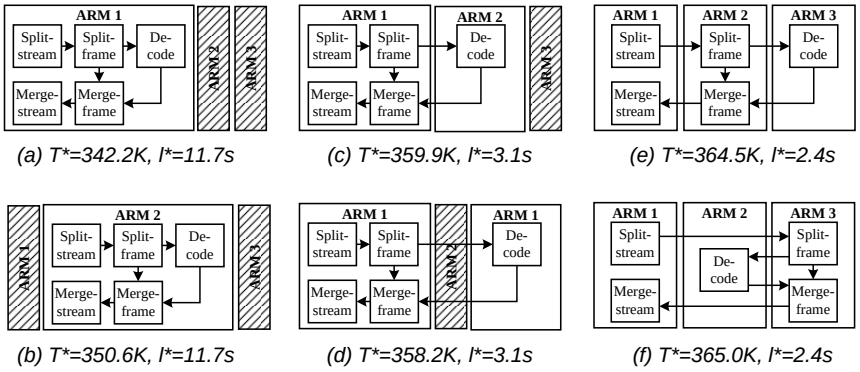


Fig. 14 Worst-case latency versus worst-case peak temperature for similar bindings but different placements, of an MJPEG decoder evaluated on MPARM platform [50].

sons for these differences. First, several operators in the formal performance analysis do not yield tight bounds. Second, the simulation of a complex system in general cannot determine the actual worst-case and best-case behavior. The simulations on the system level do not use exhaustive test patterns and do not cover all possible corner cases in the interference through joint resources.

Moreover, to illustrate the connection between the worst-case chip temperature and worst-case latency, we represent eight selected mapping configurations of the MJPEG decoder application together with their worst-case chip temperature and worst-case latency calculated in MPA. Interesting here is the effect of the the physical placement that cannot be ignored anymore. So even if the mapping is already defined, the system designer might still optimize the system (i.e., reduce the temperature) by selecting an appropriate physical placement. This is highlighted by solution pairs where only the placement of the processing components has changed but temperature differences of 8K can still appear [75].

Finally, the DOL framework itself is evaluated in terms of code size of the prototype implementation. The DOL design flow and the associated tools are implemented in Java. To give an indication about the size of the implementation, Table 4

shows the code size of different parts of the design flow (excluding the plug-ins for design space exploration and thermal analysis). One can see that apart from the tool-internal representations of the system specification, the largest part is the MPA code generator for performance analysis. The software synthesizers and the monitoring for the MPA model calibration are comparatively small. Similar observations can be made for other design flows, as well.

Table 4 Java code size of different parts of the DOL design flow.

part of design flow	**lines of code**
DOL representation of system specifications	6200
functional simulation generator	4100
MPARM code generator	2100
MPA Matlab code generator	5000
log-file analysis of functional and timed simulation	1200

5 Concluding Remarks

The mapping of process networks onto multi-processor systems requires a systematic and automated design methodology. This chapter provides an overview over different existing methods and tools, which are all starting from a general Kahn process network (KPN) model of computation and are implementing the established Y-chart approach. Due to fundamental properties of the Kahn model, many problems in the design process can be solved in an automated manner. Thus, the system specification, synthesis, performance analysis, and design space exploration can be implemented in a fully automated way.

After an overview over all these activities, this chapter provides a practical illustration of their implementation in the distributed operation layer (DOL) framework. The design steps followed by DOL are somewhat common to all the KPN flows. What is typical to the DOL framework is the embedding of an accurate formal performance analysis model into the design flow. This presents a clear advantage over the standard simulation-based approaches employed for performance analysis, which typically take more time to execute than a formal model and cannot offer guarantees for (hard) real-time signal-processing applications, due to the incomplete coverage of the design space.

Another key point is the need for a scalable design flow which allows to design large and complex MPSOC systems, which can clearly be noticed from Table 2. As soon as we are faced with more complex MPSoCs, this forthcoming difficulty needs to be considered in all steps of the design trajectory. In particular, it will have an impact at the system-level, where basic design decisions are taken. In this sense, the Kahn model and design methods based on it are promising candidates due to the

modular system specification. It offers a great potential for compositional (and fast) performance analysis and design space exploration. By taking a closer look at the DOL framework, it can be observed that it features a specification format that can easily be scaled (i.e. provided by the XML and C basis), it includes a compositional formal performance analysis in the design, and the optimization is done with the support of modular tools such as EXPO and PISA. These features provide the basis for scalable mappings and mapping optimizations.

References

1. B. Kienhuis, E. Deprettere, K. Vissers, P. van der Wolf, in *Proc. Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)* (Washington, DC, USA, 1997), pp. 338–349
2. D. Densmore, A. Sangiovanni-Vincentelli, R. Passerone, IEEE Design & Test of Computers **23**(5), 359 (2006)
3. G. Kahn, in *Proc. IFIP Congress* (Stockholm, Sweden, 1974), pp. 471–475
4. Ptolemy Web Site. http://ptolemy.eecs.berkeley.edu
5. F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Computer **36**(4), 45 (2003). DOI http://dx.doi.org/10.1109/MC.2003.1193228
6. MathWorks Real-Time Workshop. http://www.mathworks.com/products/rtw/
7. NI LabVIEW Microprocessor SDK. http://www.ni.com/labview/microprocessor_sdk.htm
8. J. Keinert, M. Streubhr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, M. Meredith, ACM Trans. on Design Automation of Electronic Systems **14**(1), 1:1 (2009)
9. S. Ha, S. Kim, C. Lee, , Y. Yi, S. Kwon, Y.P. Joo, ACM Trans. on Design Automation of Electronic Systems **12**(3), 1 (2007)
10. J. Falk, J. Keinert, C. Haubelt, J. Teich, C. Zebelein, *Integrated modeling using finite state machines and dataflow graphs*. second edition (Springer, 2012)
11. A.D. Pimentel, C. Erbas, S. Polstra, IEEE Trans. on Computers **55**(2), 99 (2006)
12. L. Thiele, I. Bacivarov, W. Haid, K. Huang, in *Proc. Int'l Conf. on Application of Concurrency to System Design (ACSD)* (Bratislava, Slovak Republic, 2007), pp. 29–40
13. H. Nikolov, T. Stefanov, E. Deprettere, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems **27**(3), 542 (2008)
14. T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, ACM Trans. on Embedded Computing Systems **5**(2), 281 (2006)
15. A. Kumar, S. Fernando, Y. Ha, B. Mesman, H. Corporaal, ACM Trans. on Design Automation of Electronic Systems **31**(3), 40:1 (2008)
16. S.S. Bhattacharyya, G. Brebner, J.W. Janneck, J. Eker, C. von Platen, M. Mattavelli, M. Raulet, in *First Swedish Workshop on Multi-Core Computing (MCC)* (Uppsala, Sweden, 2008)
17. S.A. Edwards, O. Tardieu, IEEE Trans. on VLSI Systems **14**(8), 854 (2006)
18. M.I. Gordon, W. Thies, S. Amarasinghe, in *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, USA, 2006), pp. 151–162
19. B. Kienhuis, E. Rijpkema, E. Deprettere, in *Proc. of the Int'l Workshop on Hardware/Software Codesign (CODES)* (San Diego, CA, USA, 2000), pp. 13–17
20. S. Verdoolaege, H. Nikolov, T. Stefanov, EURASIP Journal on Embedded Systems **2007** (2007)
21. A.D. Pimentel, Int. J. Embedded Systems **3**(3), 181 (2008)
22. J.W. Janneck, I.D. Miller, D.B. Parlour, G. Roquier, M. Wipliez, M. Raulet, in *IEEE Workshop on Signal Processing Systems (SiPS)* (Washington, D.C., USA, 2008), pp. 287–292
23. N. Vasudevan, S.A. Edwards, in *Proc. ACM Symposium on Applied Computing (SAC)* (Honolulu, HI, USA, 2009), pp. 1626–1631

24. D. Gelernter, N. Carriero, Commun. ACM **35**(2), 97 (1992)
25. T.M. Parks, Bounded Scheduling of Process Networks. Ph.D. thesis, University of California, Berkeley (1995)
26. IBM SDK for Multicore Acceleration. http://www-128.ibm.com/developerworks/power/cell/
27. K. Huang, I. Bacivarov, J. Liu, W. Haid, in *IEEE Symposium on Industrial Embedded Systems (SIES)* (IEEE, Ecole Polytechnique Fdrale de Lausanne, Switzerland, 2009), pp. 74–81
28. M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, J.M. Drake Moyano, in *Proc. Euromicro Conference on Real-Time Systems* (Delft, The Netherlands, 2001), pp. 125–134
29. S. Chakraborty, S. Künzli, L. Thiele, in *Proc. Design, Automation and Test in Europe (DATE)* (Munich, Germany, 2003), pp. 190–195
30. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, R. Ernst, IEE Proceedings Computers and Digital Techniques **152**(2), 148 (2005)
31. R. Alur, D.L. Dill, Theoretical Computer Science **126**(2), 183 (1994)
32. M. Hendriks, M. Verhoef, in *Workshop on Parallel and Distributed Real-Time Systems* (Rhodes, Greece, 2006)
33. S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, M. González Harbour, in *Proc. Int'l Conf. on Embedded Software (EMSOFT)* (Salzburg, Austria, 2007), pp. 193–202. DOI http://doi.acm.org/10.1145/1289927.1289959
34. E. Wandeler, L. Thiele, in *Proc. Asia and South Pacific Conf. on Design Automation (ASP-DAC)* (Yokohama, Japan, 2006), pp. 479–484
35. A. Hamann, R. Racu, R. Ernst, in *Proc. Real Time and Embedded Technology and Applications Symposium (RTAS)* (Bellevue, WA, United States, 2007), pp. 269–280
36. S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, H. Meyr, in *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Salzburg, Austria, 2007), pp. 75–80
37. I. Bacivarov, A. Bouchhima, S. Yoo, A.A. Jerraya, International Journal of Embedded Systems (IJES) **1**(1/2), 103 (2005). DOI http://dx.doi.org/10.1504/IJES.2005.008812
38. S. Schliecker, S. Stein, R. Ernst, in *Proc. Design, Automation and Test in Europe (DATE)* (2007), pp. 273–278
39. S. Künzli, A. Hamann, R. Ernst, L. Thiele, in *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)* (Salzburg, Austria, 2007), pp. 63–68
40. S. Künzli, F. Poletti, L. Benini, L. Thiele, in *Proc. Design, Automation and Test in Europe (DATE)* (2006), pp. 236–241
41. M. Gries, Integration, the VLSI Journal **38**(2), 131 (2004)
42. S. Ha, H. Oh, *Decidable dataflow models for signal processing: Synchronous dataflow and its extensions*. second edition (Springer, 2012)
43. K. Huang, W. Haid, I. Bacivarov, M. Keller, L. Thiele, ACM Transactions in Embedded Computing Systems (TECS) (2012)
44. Distributed application layer. URL http://www.tik.ee.ethz.ch/ euretile
45. M. Geilen, T. Basten, *Kahn process networks and a reactive extension*. second edition (Springer, 2012)
46. E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.Y. Brunel, W.M. Kruijtzer, P. Lieverse, K.A. Vissers, in *Proc. Design Automation Conference (DAC)* (Los Angeles, CA, USA, 2000), pp. 402–405
47. S.A. Edwards, N. Vadudevan, O. Tardieu, in *Proc. Design, Automation and Test in Europe (DATE)* (Munich, Germany, 2008), pp. 1498–1503
48. D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, K. Yazawa, IEEE Journal of Solid-State Circuits **41**(1), 179 (2006)
49. P.S. Paolucci, A.A. Jerraya, R. Leupers, L. Thiele, P. Vicini, in *Proc. Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (Seoul, South Korea, 2006), pp. 167–172

50. L. Benini, D. Bertozzi, B. Alessandro, F. Menichelli, M. Olivieri, The Journal of VLSI Signal Processing **41**, 169 (2005). DOI doi:10.1007/s11265-005-6648-1
51. T.G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (IEEE Computer Society, Washington, DC, USA, 2010), SC '10, pp. 1–11. DOI 10.1109/SC.2010.53. URL http://dx.doi.org/10.1109/SC.2010.53
52. RTEMS Home Page. http://www.rtems.com
53. W. Haid, L. Schor, K. Huang, I. Bacivarov, L. Thiele, in *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)* (Grenoble, France, 2009), pp. 35–44
54. S. Edwards, L. Lavagno, E.A. Lee, A. Sangiovanni-Vincentelli, Proceedings of the IEEE **85**(3), 366 (1997)
55. H. Kopetz, G. Bauer, Proceedings of the IEEE **91**(1), 112 (2003)
56. T.A. Henzinger, B. Horowitz, C.M. Kirsch, Proceedings of the IEEE **91**(1), 84 (2003)
57. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, R. De Simone, Proceedings of the IEEE **91**(1), 64 (2003)
58. E. Wandeler, L. Thiele, M. Verhoef, P. Lieverse, Int'l Journal on Software Tools for Technology Transfer (STTT) **8**(6), 649 (2006)
59. L. Thiele, S. Chakraborty, M. Naedele, in *Proc. Int'l Symposium on Circuits and Systems (ISCAS)*, vol. 4 (Geneva, Switzerland, 2000), vol. 4, pp. 101–104
60. R.L. Cruz, IEEE Trans. Inf. Theory **37**(1), 114 (1991)
61. J.Y. Le Boudec, P. Thiran, *Network Calculus — A Theory of Deterministic Queuing Systems for the Internet*, *Lecture Notes in Computer Science*, vol. 2050 (Springer Verlag, 2001)
62. E. Wandeler, L. Thiele. Real-Time Calculus (RTC) Toolbox. http://www.mpa.ethz.ch/Rtctoolbox (2006). URL http://www.mpa.ethz.ch/Rtctoolbox
63. W. Haid, M. Keller, K. Huang, I. Bacivarov, L. Thiele, in *Proc. Int'l Conf. on Systems, Architectures, Modeling and Simulation (IC-SAMOS)* (Samos, Greece, 2009), pp. 92–99
64. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, ACM Trans. on Embedded Computing Systems **7**(3), 36:1 (2008)
65. A.D. Pimentel, M. Thompson, S. Polstra, C. Erbas, Journal of Signal Processing Systems **50**(2), 99 (2008)
66. D. Rai, H. Yang, I. Bacivarov, J.J. Chen, L. Thiele, (DATE11, Grenoble, France, 2011)
67. L. Schor, I. Bacivarov, H. Yang, L. Thiele, in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (IEEE Computer, Beijing, China, 2012)
68. L. Thiele, L. Schor, H. Yang, I. Bacivarov, in *Proc. Design Automation Conference (DAC)* (ACM, San Diego, California, USA, 2011), pp. 268 – 273
69. E. Zitzler, L. Thiele, IEEE Trans. on Evolutionary Computation **3**(4), 257 (1999)
70. E. Zitzler, L. Thiele, J. Bader, in *Conf. on Parallel Problem Solving From Nature (PPSN)* (Dortmund, Germany, 2008), pp. 847–858
71. L. Thiele, S. Chakraborty, M. Gries, S. Künzli, in *Proc. Design Automation Conference (DAC)* (New Orleans, LA, USA, 2002), pp. 880–885
72. E. Zitzler, M. Laumanns, L. Thiele, in *Proc. Evolutionary Methods for Design, Optimisation, and Control (EUROGEN)* (Athens, Greece, 2001), pp. 95–100
73. S. Bleuler, M. Laumanns, L. Thiele, E. Zitzler, in *Int'l Conf. on Evolutionary Multi-Criterion Optimization (EMO)* (Faro, Portugal, 2003), pp. 494–508
74. G.K. Wallace, IEEE Trans. on Consumer Electronics **38**(1), 18 (1992). DOI 10.1109/30.125072
75. P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, L. Huang, (CODES+ISSS'11, October 9-14, 2011, Taipei, Taiwan., 2011)