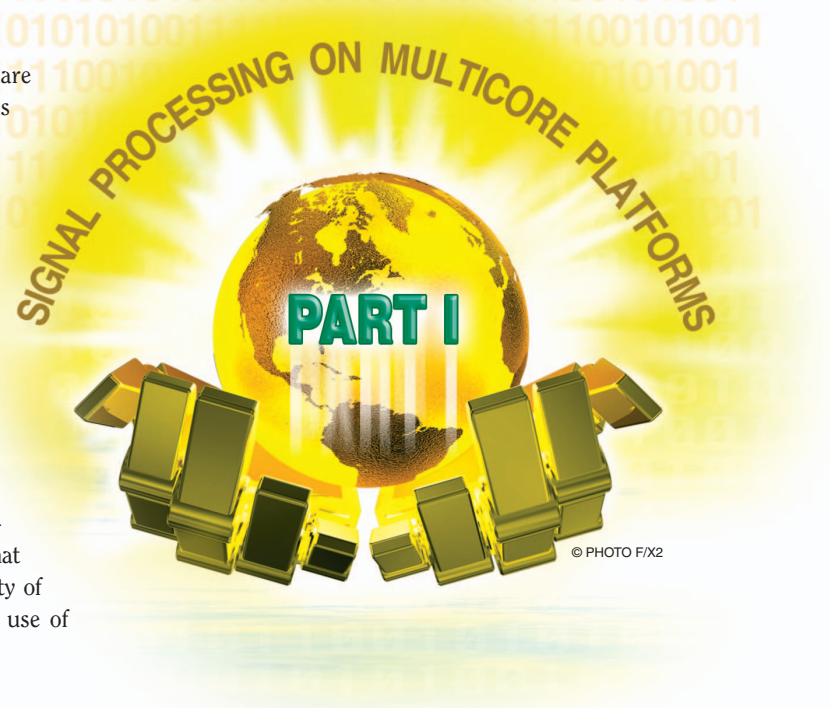
Wolfgang Haid, Kai Huang, Iuliana Bacivarov, and Lothar Thiele

Multiprocessor SoC Software Design Flows

A focus on Kahn process networks

Typical design flows supporting the software development for multiprocessor systems are based on a board support package and high-level programming interfaces. These software design flows fail to support critical design activities, such as design space exploration or software synthesis. One can observe, however, that design flows based on a formal model of computation can overcome these limitations. In this article, we analyze the major challenges in multiprocessor software development and present a taxonomy of software design flows based on this analysis. Afterwards, we focus on design flows based on the Kahn process network (KPN) model of computation and elaborate on corresponding design automation techniques. We argue that the productivity of software developers and the quality of designs could be considerably increased by making use of these techniques.



INTRODUCTION

For many emerging signal processing applications, Teraflop performance is required: High-quality multimedia processing in consumer electronics, software defined radio in communications systems, or real-time diagnostics in medical systems are typical examples. A frequent choice for digital signal processing systems will be heterogeneous multiprocessor systems-on-chip (MPSoCs) because of their computational power, programmability, and low power dissipation.

Software development plays a central role in handling the increasing complexity of applications implemented on MPSoCs. Productively programming heterogeneous MPSoCs requires support for concurrency, timing, heterogeneity, scalability, and hardware/software system integration. This support is only provided to a very limited degree by the traditional practice of using C/C++ and a board support package to program single and multiprocessor signal processing systems.

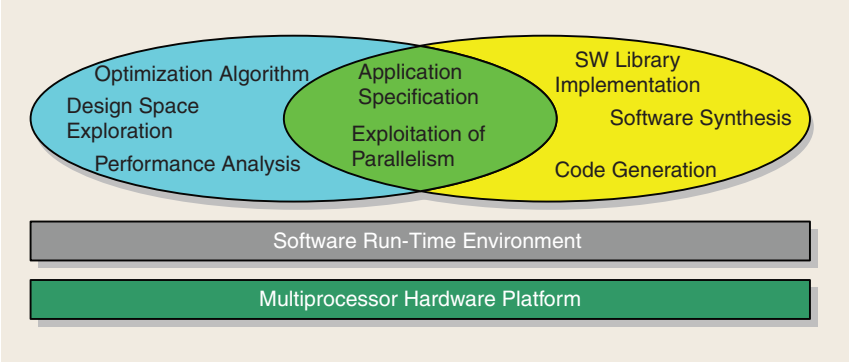
Recognizing this challenge, a variety of techniques and complete software design flows have been proposed that shift the software development for MPSoCs to higher levels of abstraction. In those design flows, applications are developed using a high-level application programming interface (API), or a model of computation. These approaches attempt to assist software

Digital Object Identifier 10.1109/MSP.2009.934111

1053-5888/09/$26.00©2009IEEE

developers in the necessary high-level design decisions and allow automating certain steps in the design flow.

One aim of this article is to review current MPSoC software design flows and classify them based on the associated challenges. To this end, we first partition the software development for MPSoCs into three phases, each targeting a certain design challenge. Afterwards, we classify design flows based on their support for tackling these challenges into flows based on a board support package, an API, or a model of computation. Based on a comparison of the three approaches, we argue that developers of signal processing applications could profit from using design flows based on a model of computation instead of a board support package or API. Later, we concentrate on the KPN model of computation [1]. Being a model of computation that is particularly well suited for signal processing applications executing on MPSoC, we survey the state-of-the-art of several design flows based on KPNs. The goal is to show concrete techniques used in these design flows, the resulting advantages, but also some disadvantages compared to traditional design flows.



[FIG1] **Elements of MPSoC software design flow and the associated challenges.**

The focus of this article is software design flows that aid software developers in implementing a parallel application on a given hardware platform. We focus on the principles and techniques underlying different software design flows rather than on providing a survey of tools. For an extensive survey of system-level design tools including software design flows, we refer to [2]. Finally, note that some of the presented principles and techniques also apply to design flows for behavioral (system-level) register transfer level (RTL) synthesis as well as hardware/software (HW/SW) codesign.

THE MPSoC SOFTWARE DESIGN FLOW

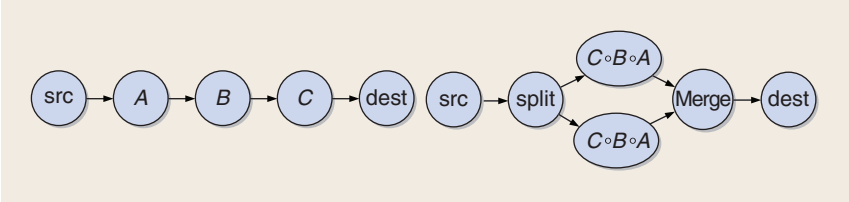
Unlike single-processor systems, the behavior of which largely depends on a single execution unit, MPSoCs have a much more complex, often nonintuitive behavior due to multiple execution units and their interaction. Developing software for MPSoCs is thus a challenging task that requires more than just writing a parallel program. Software developers need to consider how to parallelize applications, how to map parallel parts of an application to processors, or how to interface hardwired accelerators. These choices heavily influence the predictable execution, performance, and energy consumption of the final system.

The aim of an MPSoC software design flow is to aid a designer in implementing an application on a given hardware platform. Practically, this includes all the activities starting from application development to the generation of the code executing on top of the run-time environment on the target platform. We deliberately exclude the implementation of a run-time environment as well as compilation from our discussion.

The phases of a generic MPSoC software design flow are illustrated in Figure 1: application specification, software synthesis, and design space exploration.

The goal of application specification is to express the coarse-grain data and functional parallelism in the application. Fine-grained word or instruction-level parallelism can be effectively handled by today's compilers. Data parallelism can be leveraged by independently processing data in parallel while functional parallelism can be leveraged by splitting up functions into several stages that are computed in a parallel or pipelined fashion (see Figure 2). Introducing concurrency inevitably leads to communication that needs to be carefully balanced with the parallel computation. To be able to explore the resulting tradeoffs, an application specification should be modular such that different configurations can be generated without much coding effort. Besides modularity, also scalability and platform independence are highly desirable properties of an application specification that allow to easily adjust applications to the size of the problem and different platforms.

The goal of software synthesis is to actually implement a given application specification on a target platform. During this phase of refinement, the pitfalls of parallel programming, such as deadlocks, starvation, and data races need to be handled. Typically, software synthesis is based on the board support package of the target platform and existing software libraries that implement high-level APIs. The challenge is to



[FIG2] **Two functionally equivalent KPNs computing dest = $C(B(A(\text{src}))$. The left one exhibits functional parallelism while the right one exhibits data parallelism. Which configuration is more suitable depends on the target platform. $C \circ B \circ A$ denotes the concatenation of C, B, and A.**

implement an application efficiently in terms of run-time performance, but also code size, and memory consumption.

The goal of design space exploration is to determine an application configuration and a mapping to the target platform such that the design objectives are met. Typical objectives concern the system throughput or resource utilization that often need to be met under cost, power, or real-time constraints. Due to the complexity of MPSoCs, manually performing design space exploration is a difficult task requiring detailed knowledge about both the application and the target platform. On the other hand, even moderately sized systems have a design space that cannot be simply exhaustively searched due to too many application-platform-mapping combinations. Thus, there is a need to integrate search or optimization methods together with fast performance analysis methods into the design flow. To enable fast performance analysis, however, software needs to be constructed using predictable system-level concepts in terms of communication and scheduling schemes, rather than a plethora of individual concepts at the subsystem level. Otherwise, one has to resort to simulation or measurements for performance analysis that are prohibitively slow for (early) design space exploration.

TAXONOMY OF MPSoC SOFTWARE DESIGN FLOWS

A taxonomy of design flows can be derived by comparing their support for solving the challenges described above. Specifically, one can observe that the abstraction level of the programming interface used for software development determines to which extent the different activities in the design flow can be supported. Thus, we propose to distinguish between three classes of design flows based on the employed abstraction level. The three classes are board support package (BSP)-based design flows that provide little support for automating the software design flow, API-based design flows that support application specification and software synthesis, and model of computation (MoC)-based design flows that also support design space exploration and performance analysis. To underpin this taxonomy, we first give a definition of the three approaches and then comment on their differences.

BSP-, API-, AND MoC-BASED DESIGN FLOWS

BSP-based design flows are currently the most widely used alternative for programming MPSoCs. Usually, manufacturers provide a BSP for a specific MPSoC, as well as an integrated development environment that serves as the standard cockpit for platform dependent software development, testing, simulation, and debugging. Examples are Texas Instruments' Code Composer Studio, the Cell SDK for the Sony/Toshiba/IBM Cell Broadband Engine, or the Compute Unified Device Architecture (CUDA) IDE of NVIDIA.

BSP-based design flows do not explicitly separate the three design phases. This results in a large flexibility to implement a system at the expense of a high development effort and missing support for automated design space exploration and software synthesis.

API-based design flows are based on a platform independent high-level API. This API defines how different components of an application execute concurrently and how these components communicate. An API can, for instance, include functions for synchronous and asynchronous communication, point-to-point and multicast communication, synchronization routines, or mechanisms for mutual exclusion and atomicity of code blocks. More formally, Skillicorn and Talia [3] classify APIs with respect to the key concepts of parallel computing, namely parallelism, decomposition, mapping, communication, and synchronization. Examples of APIs for programming MPSoCs are the widely used message passing interface (MPI) and OpenMP libraries as well as TTL [4] or Multiflex [5].

API-based design flows reduce the development effort by relieving the programmer from developing platform specific low-level code (assuming that the API has already been ported to the target platform). Note that this does not necessarily incur a performance penalty: An API routine for communication between two parallel processes, for instance, can be refined differently depending on whether the processes are located on the same processor or on different ones [4], [5]. While API-based design flows considerably simplify software synthesis, they still do not provide support for automated design space exploration.

MoC-based design flows restrict applications to an MoC that defines the semantics of concurrently executing components and their interaction. Restricting to an MoC helps in tackling many of the challenges mentioned above by completely avoiding certain problems (by restricting the way how applications are constructed), by increasing predictability (by exploiting a-priori knowledge about the behavior of the application), or by detecting problems early in the design flow (by static analysis). Examples of MoCs are (Kahn) process networks, the discrete event model, finite state machines, synchronous/reactive models, or combinations thereof. Lee and Sangiovanni-Vincentelli [6] present a framework for comparing MoCs. An overview of applying MoCs in system design is presented in [7].

MoC-based design flows separate all three design phases and allow the automation of design space exploration due to the restricted application behavior. Obviously, these design flows are suited for a narrower scope of applications compared to BSP- or API-based design flows because an MoC rules out certain kinds of algorithms (or at least makes them very cumbersome to implement). One can observe, however, that many signal processing and embedded applications are only mildly restricted when choosing a suitable MoC. For these applications, the benefits provided by MoC-based design flows outweigh the restrictions.

An MoC frequently used for signal processing applications is the KPN model [1] and its ramifications, such as dataflow, synchronous dataflow, and cyclo-static dataflow [8]. In the section "KPN Design Flows," we focus on the KPN model due to its relevance for signal processing applications executing on MPSoCs.

COMPARISON OF DESIGN FLOWS

A comparison of the design flows is presented in Table 1. The table shows how raising the level of abstraction influences the

[TABLE 1] COMPARISON OF APPROACHES FOR MULTIPROCESSOR SOFTWARE DEVELOPMENT.

	BSP	API	MoC
PROGRAMMING INTERFACE	PLATFORM SPECIFIC	PLATFORM INDEPENDENT	PLATFORM INDEPENDENT, FORMAL SEMANTICS
APPLICATION SCOPE	ANY	MANY	FEW
PARALLELISM	IMPLICIT	EXPLICIT	EXPLICIT
POTENTIAL FOR RACE CONDITIONS, DEADLOCKS, ETC.	HIGH	MIDDLE	LOW
SOFTWARE SYNTHESIS	MANUAL	LIBRARY-BASED	AUTOMATED
TASK-LEVEL DESIGN SPACE EXPLORATION	MANUAL	MANUAL	INTERACTIVE, AUTOMATED
PERFORMANCE ANALYSIS (BESIDES MEASUREMENT)	SIMULATION	SIMULATION	SIMULATION, ANALYTIC

scope of a design flow, application specification, software synthesis, and design space exploration. In BSP-based design flows, application developers have many freedoms in tailoring an application to a platform. This can result in highly efficient implementations when the developer has the required knowledge, experience, and time to optimize the program code. On the contrary, the API- and MoC-based design flows integrate concepts of platform-based design [9] and model-driven development [10]. By raising the level of abstraction and automation, developers can concentrate on relevant aspects of their work without being immediately immersed in the details of implementation. While this results in an increased productivity, the quality of the implementation relies on optimization tools and developers sacrifice some control over the final implementation.

From another perspective, the differences between the three classes of design flows are exemplified in Listing 1. It shows how a chosen software development approach and level of abstraction, respectively, directly influence the way software developers write applications. Specifically, Listing 1 shows how to implement communication between tasks running on different processors using the three approaches. The example is for the Sony/Toshiba/IBM Cell Broadband Engine. In Listing 1, the BSP is the Cell SDK, the API is the MPI library, and the MoC is KPN.

In the BSP-based approach, low-level routines or macros provided by the BSP are used to setup a memory flow controller to carry out the data transfer. The programmer is required to have a detailed understanding of the target platform and needs to deal with the low-level details of the implementation. Much time is thus often spent for creating a functionally correct implementation, rather than for optimizing an implementation.

In the API-based approach using MPI, portable MPI routines are used to read data. In contrast to the BSP-based implementation, the program code is platform independent and could be easily ported to another platform. By relieving the programmer from dealing with low-level details, the API-based approach scales well to large systems. Still, the modularity of a process in MPI is limited because the *MPI_Recv* routine, for instance, contains a reference to the source process. Consequently, exploring different application configurations and mappings is tedious because it requires the modification of source code of processes.

In an MoC-based approach using the KPN model, data can only be read from a first-in first-out channel. In contrast to the other two approaches, the communication between processes is much more restricted. While this reduces flexibility, it simplifies automated analysis and design space exploration because channels are statically allocated and cannot be changed during run time. Contrary to the BSP- and API-based approach, a process is completely modular because it reads and writes data from and to (local) ports that conceptually can be connected to any other process. Thus, exploring different application topologies does not require the modification of the source code of processes but just a redefinition of the interconnection network.

Finally, it can be observed that the line between the classes is sometimes blurred when elements of more than one approach are combined in a software design flow. One possibility, for instance, is to combine the advantages of the three approaches by employing an API- or MoC-based design flow to create a "baseline" implementation and manually optimize it by fine-tuning critical components and removing bottlenecks. Another frequently used option is to include elements of the API- or MoC-based approach in a BSP-based design flow. The Cell SDK, for instance, contains elements of an API-based approach in the form of the Accelerated Library Framework (ALF). Another example is the CUDA IDE featuring a thread-based programming model. The tool support for those parts of a program that adhere to this programming model resembles the support in MoC-based design flows. In the light of these combined approaches, the

LISTING 1: IMPLEMENTATION OF READ ROUTINE USING THREE DIFFERENT SOFTWARE DEVELOPMENT APPROACHES.

```
//BSP-based code (Cell SDK)
uint32_t tag_id = mfc_tag_reserve();
mfc_get(&buffer, src, size,
  tag_id, 0, 0);
mfc_write_tag_mask(1 << tag_id);
mfc_read_tag_status_all();

//API-based code (MPI)
MPI_Comm_get_attr(MPI_COMM_WORLD,
  MPI_TAG_UB,&tag_id, &i);
MPI_Recv(&buffer, 0, MPI_BYTE, src,
  tag_id,MPI_COMM_WORLD, &status);

//MoC-based code (Kahn process network)
read(&buffer, port_id, size);
```

[TABLE 2] KPN DESIGN FLOWS.

DESIGN FLOW	WEB PAGE
DAEDALUS [11, 12]	HTTP://DAEDALUS.LIACS.NL
DIF [13]	HTTP://WWW.ECE.UMD.EDU/DSPCAD
DOL [14]	HTTP://WWW.TIK.EE.ETHZ.CH/~SHAPES
KOSKI [15]	NOT AVAILABLE ONLINE
MAMPS [16]	HTTP://WWW.ES.ELE.TUE.NL/MAMPS
MICROPROCESSOR SDK [17]	HTTP://WWW.NI.COM/LABVIEW/
OPENDF [18]	HTTP://OPENDF.SOURCEFORGE.NET
PTOLEMY II [19]	HTTP://PTOLEMY.EECS.BERKELEY.EDU
REAL-TIME WORKSHOP [20]	HTTP://WWW.MATHWORKS.COM
SHIM [21]	NOT AVAILABLE ONLINE
STREAMIT [22]	HTTP://WWW.CAG.LCS.MIT.EDU/STREAMIT

presented taxonomy can help software developers to decide which approach to use for the different parts of an application.

KPN DESIGN FLOWS

So far, we have discussed the challenges of MPSoC software design and argued that MoC-based design flows are well suited to handle them. In this section, we focus on a particular MoC, namely the KPN [1], and techniques that are applied in KPN design flows. Table 2 shows an overview of KPN design flows, many of which are publicly available.

The KPN model has become a popular model of computation for parallel applications executing on MPSoCs because it is a good match for many applications as well as hardware platforms. With respect to applications, the KPN model allows making explicit the data and functional parallelism in an application. Being conceptually similar to signal flow graphs, many signal processing applications can be naturally modeled using the KPN model. With respect to hardware platforms, the KPN model is determinate, which allows executing KPN applications in an untimed (asynchronous) fashion [8]. Requiring no global coordination or synchronization, KPN applications can be efficiently executed on a wide range of platforms with different processor, interconnect, and memory configurations. The design flows listed in Table 2, for instance, have been applied for the following platforms (among others): Atmel DIOPSIS 940 (DOL), Sony/Toshiba/IBM Cell Broadband Engine (Daedalus, SHIM, StreamIt), Intel Quad-Core Xeon (SHIM), multi-MicroBlaze implemented on a Xilinx Virtex-II Pro FPGA (Daedalus, MAMPS), and multi-NIOS implemented on an Altera Stratix-II FPGA (Koski).

The aim of this section is to highlight the advantages of using KPN design flows compared to BSP- or API-based flows, rather than to compare KPN design flows against each other.

[TABLE 3] LINES OF CODE IN THE HIGH-LEVEL SPECIFICATION LANGUAGE (LOC-SL) AND OF A PLATFORM DEPENDENT IMPLEMENTATION IN C/C++ (LOC-C) BASED ON A BSP. THE LOC-C FOR DOL AND OPENDF REFER TO AUTOMATICALLY SYNTHESIZED IMPLEMENTATIONS.

DESIGN FLOW	APPLICATION	LOC-SL	LOC-C
DOL [14]	MPEG2 DECODER	4000	6200
OPENDF [18]	MPEG4 SP DECODER	3400	10400
STREAMIT [22]	MPEG2 DECODER	3200	6835

Specifically, we survey techniques to deal with the challenges encountered in the development of multiprocessor signal processing applications. Where appropriate, we refer to implementations in concrete design flows.

APPLICATION SPECIFICATION

The KPN model can be seen as a coordination model [23], which views the programming of a distributed parallel system as the combination of two distinct activities: the computing part comprising a number of processes involved in manipulating data and a coordination part for communication and cooperation between the processes. KPN applications are usually specified in a way to reflect this model.

Two different approaches can be distinguished, namely specification using a host and a coordination language and specification using a domain-specific language. When using a host and a coordination language, 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). When using a domain-specific language, computation and coordination are expressed in a single language that provides constructs for both parts. 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 executes independently from other processes in the system except for the data dependencies that are made explicit by channels. Referring to the challenges mentioned in the section "The MPSoC Software Design Flow," these properties allow for modular, scalable, and platform-independent application specifications. Programming an application using these languages is thus considerably different compared to the BSP- or API-based approach. The KPN-based approach lets application developers focus on expressing an application as processes and channels while much of the time in a BSP- or API-based approach is spent on the technical implementation of these system-level concepts.

To illustrate this difference, Table 3 compares the lines of the source code required to specify an application using a KPN specification language compared to the lines of C/C++ code required for the actual system implementation. Taking lines of code as an indicator for the time to write that code, there is a clear gain when using an MoC-based approach compared to a BSP- or API-based approach.

SOFTWARE SYNTHESIS

The MoC-based design 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 one hand and achieve the desired performance of the complete HW/SW system on the other. We use the term software synthesis to denote the translation from a high-level KPN specification to a platform specific implementation in a sequential language, such as C or C++. This

involves the generation of the code implementing the KPN topology, the generation of the wrapper code that implements processes in a given run-time environment, and the generation of the code for bootstrapping the application. If a domain-specific language, rather than C/C++, is used for specifying process behavior, software synthesis also involves the source-to-source code transformation from the specification language to C/C++. In an even broader sense, software synthesis might also deal with further aspects, such as deriving a static schedule or minimizing the memory footprint [24]. Software synthesis is followed by compilation that generates the software binaries for the different processors in an MPSoC. Note that compilation often takes longer than the code generation if software synthesis is automated. Table 4 reports the duration of software synthesis for several design flows.

The automated synthesis of program code is presumably the biggest benefit of using a KPN design flow or an MoC-based design flow, in general. Being able to implement different application configurations by just modifying the system-level application specification and automatically synthesize the corresponding source code lets application developers (and design automation tools) quickly evaluate different configurations. In the context of HW/SW codesign, the Artemis [11], ESPAM [12], Koski [15], MAMPS [16], OpenDF [18], and SHIM [21] design flows even allow to synthesize software and hardware of MPSoCs based on KPN application specifications for implementation on field-programmable gate arrays (FPGAs).

DESIGN SPACE EXPLORATION

Software designers for MPSoCs face a large design space. At several points in the design flow and at various levels of abstraction, they need to decide between design alternatives. In software design, this is mainly done at the task level where alternatives may consist of different application partitions, different mappings of software tasks to resources, and different scheduling policies implemented on shared resources. In a broader sense, software designers might even need to deal with architectural aspects, such as clock frequencies in a system that supports dynamic voltage scaling or different system configurations in a dynamically reconfigurable system. In BSP- or API-based design flows, the application developer needs to take the according decisions and create the corresponding implementation. In KPN design flows, however, design space exploration can be automated due to the separate specification of application and target platform, the modular structure of KPNs, and automated software synthesis. The goal is to find an optimal design or (in the case of a multiobjective optimization problem) a set of Pareto-optimal designs amongst that the designer chooses the final implementation.

Most of the optimization problems encountered during design space exploration are NP-hard. Optimally solving these problems using exact methods is usually not feasible, therefore algorithms that approximate the optimal solution are used. Depending on the optimization goal and the problem setting, many heuristic and randomized methods have been proposed. Frequent choices are greedy algorithms, evolutionary algorithms, or simulated annealing. Gries [25] presents an overview of automated design space exploration and performance analysis in different design flows.

[TABLE 4] DURATION OF SOFTWARE SYNTHESIS FOR MPSoCS. IN CASE OF DAEDALUS, THE TIME FOR CREATING THE RTL CODE OF THE MPSoC IS INCLUDED.

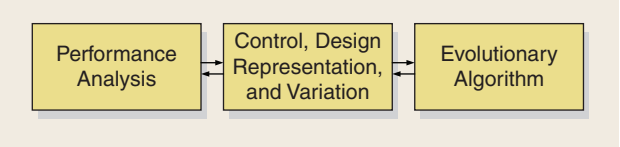
DESIGN FLOW	APPLICATION	TIME FOR SYNTHESIS
DAEDALUS [12]	MOTION JPEG	150 S
DOL [14]	MPEG2 DECODER	4 S
MAMPS [16]	JPEG-H263	1 S

As a typical example for design space exploration in a KPN design flow, we consider the Sesame design space exploration framework of Artemis [11]. This framework uses multiobjective evolutionary algorithms and high-level models for performance analysis. Specifically, using evolutionary algorithms as an optimization method allows structuring the design space exploration framework into three distinct modules, as shown in Figure 3. The design space exploration framework of DOL [14] is structured in the same way, for instance.

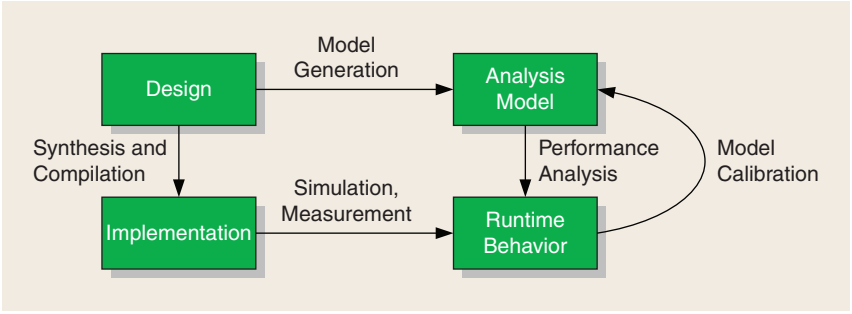
The control module at the core of the framework coordinates the design space exploration process, maintains a population of possible designs, and implements the variation of designs according to the working principle of evolutionary algorithms. For performance analysis, a separate module is used that either implements a simple analytic model (in early design phases) or trace-based simulation (in later design phases). The evolutionary algorithm itself is basically problem-independent and can be freely chosen. In the case of Sesame, SPEA2 [26] is used. The interface between the modules is based on the PISA protocol [27].

PERFORMANCE ANALYSIS

Design space exploration as well as final system verification require performance analysis to determine performance metrics of a system under consideration, such as system throughput, resource utilization, or energy consumption. In the context of MPSoCs, this poses a major challenge due to multiple hardware resources, the distributed execution of the application, and the interaction of computation and communication on shared resources. Facing this challenge, many performance analysis methods for MPSoCs have been proposed that differ in scope, accuracy, evaluation time, and setup effort. In general, one can distinguish between simulation-based and analytic methods.



[FIG3] Design space exploration framework based on multiobjective evolutionary algorithms.



[FIG4] Integration of performance analysis into the MPSoC design flow.

A wide range of simulators at different levels of abstractions is available today. The most accurate but also slowest simulators are cycle-accurate simulators. Instruction-accurate simulators (also referred to as instruction-set simulators or virtual platforms) provide a good tradeoff between speed and accuracy that allows to model and simulate entire MPSoCs. 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. At even higher levels of abstraction, simulation is sometimes mixed with analytic methods like in trace-based simulation in the Artemis design flow [11], for instance.

The two main drawbacks of simulation are that the execution time is usually rather long and that the results are only valid for a particular run of the simulation. Therefore, analytic methods are used as an alternative for faster evaluation and for making general statements about the system behavior. On the other hand, analytic methods usually suffer from a limited modeling scope and are thus only suited for systems in a limited number of domains. Examples are worst-case/best-case analysis for (hard) real-time systems or probabilistic analysis for networking systems.

An example for a design flow that integrates worst-case/best-case analysis is the DOL [14] design flow. Figure 4 illustrates the basic concept of the DOL that is targeted at real-time signal processing applications. Like in most other design flows, performance analysis can be done by simulation that basically just requires the generation of the software binaries by software synthesis and compilation. Contrary to other design flows, a detailed analytic model for real-time analysis of the system can be generated as well. Specifically, modular performance analysis [28] is used. Design space exploration and verification of real-time properties of the system can be carried out efficiently based on this model. This is enabled by the modularity of the KPN application that allows to quantitatively characterize the single components in isolation. The interaction of components that determines the final system behavior is analyzed by appropriately composing the components in the analysis model.

LIMITATIONS OF MoC-BASED DESIGN FLOWS

In the previous sections, we have seen the benefits of using an MoC-based software design flow. Despite these advantages, MoC-based design flows are not widely used in practice and the vast majority of software projects is still relying either on a BSP- or an API-based approach. We summarize some of the reasons that presumably inhibit the adoption of MoC-based design flows on a bigger scale.

First, switching from a BSP-based or an API-based design flow to an MoC-based design flow constitutes a disruptive paradigm shift rather than a smooth transition between design flows and thus incurs considerable costs [10]. This includes the up front costs of training staff, purchasing new tools, and adopting existing IPs. Also, the recurring costs for operating an MoC-based design flow might be higher because of a smaller knowledge base, less support by hardware vendors, and missing industry-wide standards.

Second, the MoC-based approach suffers from a limited modeling scope. Certain parts of a system are thus usually very cumbersome to model in an MoC, leading to inefficient implementations. In essence, MoC-based design flows can hardly replace BSP-based or API-based design flows completely. Maintaining several design flows, however, might turn out to be uneconomic in many cases.

Third, the potential of MoC-based approaches can only be brought to bear when the according optimization and synthesis techniques are available. The question is to which degree the promising techniques proposed in the literature are applicable to the design of commercial applications.

Finally, the software tool support for application specification, software synthesis, and design space exploration is relatively immature. Due to the novelty and complexity of the existing tools, they have a steep learning curve and their usage is nonintuitive to practitioners that are used to traditional software development approaches. In addition, the interoperability of tools is currently still severely limited due to missing standards. It is difficult, for instance, to use the design space exploration framework of one design flow and the software synthesis back-end from another due to a missing format for data exchange.

CONCLUSION

The current industry practice of providing just a BSP for MPSoC software development fails to support critical design activities, such as design space exploration or software synthesis. To tackle this problem, software design flows that move the software development to higher levels of abstraction are required. In this article, we presented a taxonomy of software design flows and showed their differences concerning their support for different design phases. By using a suitable design flow, the productivity of developers and the quality of designs could be considerably increased compared to the traditional BSP-based design flow. For signal processing applications, design flows based on the KPN model of computation are particularly interesting. To this end, we highlighted design automation techniques that are used in these design flows.

ACKNOWLEDGMENTS

This work was partially supported by the EU Framework Programme Projects SHAPES and COMBEST, under grant numbers 026825 and 215543.

AUTHORS

Wolfgang Haid (wolfgang.haid@tik.ee.ethz.ch) is a Ph.D. student in the Computer Engineering and Networks Laboratory of the Swiss Federal Institute of Technology, Zurich (ETH). He received his B.Sc. degree in telematics from Graz University of Technology, Austria and his M.Sc. degree in electrical engineering from ETH Zurich, in 2004 and 2006, respectively. His research interests include performance analysis and software development for MPSoC, especially for real-time applications.

Kai Huang (kai.huang@tik.ee.ethz.ch) is a Ph.D. student in the Computer Engineering and Networks Laboratory of the Swiss Federal Institute of Technology, Zurich. He received the B.Sc. degree in computer science at Fudan University, China, in 1999 and the M.Sc. degree in computer science at Leiden University, The Netherlands, in 2005. His research interests include design, analysis, and performance optimization of embedded systems.

Iuliana Bacivarov (iuliana.bacivarov@tik.ee.ethz.ch) received the electrical engineering degree in 2002 from the National Polytechnic Institute of Bucharest, Romania. In 2002–2003, she received a master's degree in microelectronics integrated systems design from the Université Joseph Fourier in Grenoble, France, as well as a master's degree in quality and reliability engineering from the National Polytechnic Institute of Bucharest. She received her Ph.D. degree in microelectronics from the National Polytechnic Institute of Grenoble in 2006. She has been a post-doctoral researcher in the Computer Engineering and Networks Laboratory of ETH Zurich since 2006. Her research interests include design, analysis, and optimization of MPSoC.

Lothar Thiele (thiele@tik.ee.ethz.ch) joined the Swiss Federal Institute of Technology, Zurich (ETH) as a full professor of computer engineering in 1994, where he currently leads the Computer Engineering and Networks Laboratory. His research interests include models, methods, and software tools for the design of embedded systems, embedded software, and bioinspired optimization techniques. He received the 1986 Dissertation Award of the Technical University of Munich, the 1987 Outstanding Young Author Award of the IEEE Circuits and Systems Society, the 1988 IEEE Browder J. Thompson Memorial Award, and the 2000–2001 IBM Faculty Partnership Award. In 2004, he joined the German Academy of Natural Scientists Leopoldina. He received the 2005 Honorary Blaise Pascal Chair of Leiden University, The Netherlands.

REFERENCES

[1] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. IFIP Congr.,* Stockholm, Sweden, Aug. 1974, pp. 471–475.

[2] D. Densmore, A. Sangiovanni-Vincentelli, and R. Passerone, "A platform-based taxonomy for ESL design," *IEEE Des. Test Comput.*, vol. 23, no. 5, pp. 359–374, May 2006.

[3] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 123–169, June 1998.

[4] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink, "Design and programming of embedded multiprocessors: An interface-centric approach," in *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Stockholm, Sweden, Sept. 2004, pp. 206–217.

[5] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagné, and G. Nicolescu, "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," *IEEE Trans. VLSI Syst.*, vol. 14, no. 7, pp. 667–680, July 2006.

[6] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.

[7] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proc. IEEE*, vol. 85, no. 3, pp. 366–390, Mar. 1997.

[8] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–799, May 1995.

[9] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Des. Test Comput.*, vol. 18, no. 6, pp. 23–33, Nov./Dec. 2001.

[10] B. Selic, "Model-driven development: Its essence and opportunities," in *Proc. IEEE Int. Symp. Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, Gyeongju, Korea, Apr. 2006, pp. 313–319.

[11] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 99–112, Feb. 2006.

[12] H. Nikolov, T. Stefanov, and E. Deprettere, "Systematic and automated multiprocessor system design, programming, and implementation," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 3, pp. 542–555, Mar. 2008.

[13] The dataflow interchange format [Online]. Available: http://www.ece.umd.edu/DSPCAD/dif/

[14] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *Proc. Int. Conf. Application of Concurrency to System Design (ACSD)*, Bratislava, Slovak Republic, July 2007, pp. 29–40.

[15] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, and T. D. Hämäläinen, "UML-based multiprocessor SoC design framework," *ACM Trans. Embedded Comput. Syst.*, vol. 5, no. 2, pp. 281–320, May 2006.

[16] A. Kumar, S. Fernando, Y. Ha, B. Mesman, and H. Corporaal, "Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 31, no. 3, pp. 40:1–40:27, July 2008.

[17] NI LabVIEW Microprocessor SDK [Online]. Available: http://www.ni.com/labview/microprocessor_sdk.htm

[18] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow programs: An MPEG-4 simple profile decoder case study," in *Proc. IEEE Workshop Signal Processing Systems (SiPS)*, Washington, DC, Oct. 2008, pp. 281–286.

[19] Ptolemy Project Home Page [Online]. Available: http://ptolemy.eecs.berkeley.edu

[20] MathWorks Real-Time Workshop [Online]. Available: http://www.mathworks.com/products/rtw/

[21] S. A. Edwards and O. Tardieu, "SHIM: A deterministic model for heterogeneous embedded systems," *IEEE Trans. VLSI Syst.*, vol. 14, no. 8, pp. 854–867, Aug. 2006.

[22] M. Drake, H. Hoffmann, R. Rabbah, and S. Amarasinghe, "MPEG-2 decoding in a stream programming language," in *Proc. Int. Parallel and Distributed Processing Symp. (IPDPS)*, Rhodes Island, Greece, Apr. 2006.

[23] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 97–107, Feb. 1992.

[24] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Norwood, MA: Kluwer, 1996.

[25] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integr. VLSI J.*, vol. 38, no. 2, pp. 131–183, Dec. 2004.

[26] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization," in *Proc. Evolutionary Methods for Design, Optimisation, and Control (EUROGEN)*, Athens, Greece, Sept. 2001, pp. 95–100.

[27] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA—A platform and programming language independent interface for search algorithms," in *Proc. Int. Conf. Evolutionary Multi-Criterion Optimization (EMO)*, Faro, Portugal, Apr. 2003, pp. 494–508.

[28] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse, "System architecture evaluation using modular performance analysis: A case study," *Int. J. Softw. Tools Technol. Transfer*, vol. 8, no. 6, pp. 649–667, Nov. 2006. [SP]