Mapping Applications to Tiled Multiprocessor Embedded Systems

Lothar Thiele Iuliana Bacivarov Wolfgang Haid Kai Huang
Swiss Federal Institute of Technology Zurich
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
{thiele, bacivarov, haid, huang}@tik.ee.ethz.ch

Abstract

Modern multiprocessor embedded systems execute a large number of tasks on shared processors and handle their complex communications on shared communication networks. Traditional methods from the HW/SW codesign or general purpose computing domain cannot be applied any more to cope with this new class of complex systems. To overcome this problem, a framework called Distributed Operation Layer (DOL) is proposed that enables the efficient execution of parallel applications on multiprocessor platforms. Two main services are offered by the DOL: system-level performance analysis and multi-objective algorithm-architecture mapping. This paper presents the basic principles of the DOL, the specification mechanisms for applications, platform and mapping as well as its internal analytic performance evaluation framework. To illustrate the presented concepts, an MPEG-2 decoder case study is presented.

1 Introduction

The complexity of applications found in modern embedded systems is steadily increasing. Video and audio codecs, modems for wireless communication, and other kinds of digital signal processing applications have performance requirements that cannot be satisfied by single processor architectures any more. The focus in embedded system design is moving away from single processor implementations towards heterogeneous multiprocessor system-on-chip (MpSoC) architectures.

While offering high scale integration, high computing power and low power consumption, MpSoC designs also face challenges:

- The intrinsic computational power of an MpSoC system should not only be used efficiently, but the time and effort to design a system containing both hardware and software must also remain acceptable. System design methods implementing applications with the latest multiprocessor architecture must harness exponential hardware resource growth in a scalable and modular way.
- MpSoC systems need to be flexible enough that the design can be re-used for different product variants or versions, and easily modified in response to market needs, user requirements, and product updates during the design and product life cycle.
- MpSoC systems are characterized by a large design space as there is a large degree of freedom in the partitioning of parallel application tasks, the allocation of concurrent hardware components, their binding to application processes, and the choice of appropriate resource allocation schemes. It is well acknowledged that the design of such complex systems requires performance evaluation and validation techniques during the whole design trajectory. Because of the overall system complexity, fast evaluation methods in an early design stage are critical for making profound design decisions.
- Embedded systems are very often real-time systems in which timeliness of the task execution is as important as the function of the task. Usually, timeliness can be easily achieved by over-dimensioning the hardware resources or reducing the application performance. But the challenge is to deliver timing guarantees without such forms of overprovisioning.

It is questionable whether the established design methodologies and tools for single processor systems are appropriate to meet these challenges. Without a disciplined design methodology, however, system designers will have to resort to ad-hoc techniques to implement concurrent applications on complex multiprocessor platforms, which is a doubtful proposition. We believe that new design methodologies and engineering practices are necessary for the design of complex MpSoC systems to achieve the productivity and efficiency that has been reached for single processor systems.

In this paper, we present the Distributed Operation Layer (DOL) as a system-level framework for solving some the above mentioned challenges. The key elements of the DOL are a programming model, a communication interface, a tailored hardware abstraction, a design space exploration

strategy, and a system-level performance analysis, which seamlessly integrates into a complete development environment for multiprocessor system design. We define a consistent interface which can be designed once and used for the whole design life cycle, e.g. application development, functional simulation, hardware-dependent software generation, cycle-true simulation and final implementation.

The contributions of this paper can be summarized as follows:

- We define abstraction models and the corresponding programming paradigms for the application and architecture, as well as a format for the mapping specification.
- We show how an analytic performance analysis strategy is integrated into the DOL framework, alleviating the designer from the difficult task to model and analyze the MpSoC system using other methods.
- For design-space exploration, we integrate a multi-objective optimization tool leveraging evolutionary algorithms.
- We present a case study to testify the developed methodology.

The rest of this paper is organized as follows. After investigating the related work in Section 2, the DOL framework is briefly introduced in Section 3, while in Section 4, the DOL programming model, the mapping and architecture specifications are presented in detail. The mapping exploration loop is described in Section 5. Section 6 presents the performance evaluation strategy while experimental results are illustrated in Section 7.

2 Related work

Application modeling: In the literature, a variety of models of computation and associated specification languages have been proposed. Prominent examples are for instance, Codesign Finite State Machine (CFSM) from the Polis project [1], Metropolis Meta-Model (MMM) language from the Metropolis project [2], the StreamIt language [8], the Tiny-SHIM [6], and Statecharts [9].

A widely used model for application modeling at the process-level is the class of Kahn process networks (KPN) [11] because of their simple communication and synchronization mechanisms. A well-known property of KPNs is that they are determinate, i.e., the functional input/output relation is independent of the timing of the processes, communication and synchronization. Several, more restrictive models have been derived from KPNs. One of these models is Synchronous Data Flow (SDF) [14, 16] and its ramifications, in which the processes of a KPN are replaced by atomic actors that can be fired when input data is available. Once an actor has been fired, it cannot stall. Another recent variation of KPN is SW/HW integration medium (SHIM) [6], the communication of which is synchronous in the sense that both sending and receiving processes must agree when data are to be transferred.

We adopt process networks as our application model, not restricting ourselves to any of the models mentioned above. We only define the semantics of the communication as point-to-point first-in first-out (FIFO) channels, and leave the implementation open for later refinements in the mapping stage. Therefore, specialized process networks semantics, such as KPN, SDF, and SHIM, can be obtained by imposing additional restrictions or semantics onto the process network.

Model specification: The syntactic specification of the application model and architecture model is also a major concern of our work. In our framework, we do not intend to create a new modeling language for the process network model. We argument that designing a new language is not the best idea, since most of existing applications are written in imperative languages like C/C++. However, we decouple the behavior of the processes from the structure of the process network. In particular, we propose an XML Schema to specify the process network, similar to Y-chart Modeling Language (YML) [18] proposed by Artemis and MoML [15] proposed in Ptolemy. Therefore, different levels of potential parallelism are clearly separated such as inter-process parallelism, i.e. thread-level or instruction-level, and intra-process concurrency.

An innovative spark of our XML Schema is the introduction of an `iterator` element which is used to describe complex repetitive models, as well in the underlying hardware platform (e.g. a massively parallel computer architecture consisting of regular tiles and repetitive communication channels) and in the application (e.g. iterated processes and communication links). In the literature, only YML [18] reports a built-in scripting support that is similar to the proposed approach.

Simulation-based approaches: Simulation-based techniques are the major approaches to validate the functionality of multiprocessor embedded systems during the design cycle and to perform a system-wide performance estimation. Polis [1] is a framework targeting control-dominated systems whose implementation is based on a micro-controller for tasks to be implemented in software and ASICs for tasks to be implemented in hardware. The Ptolemy [4] project is a flexible environment for the simulation and prototyping of heterogeneous systems, which supplies different models of computation within a single system simulations. Metropolis [2], a follow-up to the Polis project, provides a meta-model and a unified structure for simulation of multiple computation models. There are also industrial simulation tools available, such as VCC from Cadence [23] and Seamless from Mentor Graphics [19]. All these academic and industrial

simulation frameworks have in common that they allow the modeling of systems at any level of detail. However, they suffer from long run times and from a high set-up effort for each new architecture and mapping. Worst-case bounds of system properties like throughput and end-to-end delay are difficult to obtain because corner cases of the execution are difficult to identify due to the overall complexity of today's systems.

To achieve shorter run-times for simulation-based methods, approaches that combine simulation and analysis have been proposed. Lahiri et al., proposed a hybrid trace-based simulation methodology [13] for on-chip communication exploration. Künzli et al., developed a technique [12] that embedded an analytic perfromance evaluation method [22] into the SystemC based MPARM simulation tool [20], [17]. The Artemis [18] project proposed a trace-driven hardware-software cosimulation. Although these mixed methodologies can help to shorten the run-time of simulations, the problem of insufficient corner case coverage still remains.

Analysis-based approaches: In order to evaluate different platform-mapping combinations for the purpose of design space exploration, the applied methods should be modular, they should support a large variety of different resource types and resource sharing disciplines. In comparison to simulation-based methods, analytic approaches can guarantee system properties and they are characterized by a small run-time, i.e. they well scale to large MPSoC. On the other hand, the obtained bounds may be far away from the actual worst case, especially if the chosen abstractions do not cover the essential system properties. In addition to the Modular Performance Analysis (MPA), see e.g. [24], [5], that is based on Real-Time Calculus [22] there are two other prominent analytic approaches for the analysis of distributed embedded systems, namely MAST [7] and Symta/S [10]. They are both based on extensions to classical real-time scheduling theory. In this paper, we will use the MPA framework whose abstractions are adapted to the complex stream patterns as occurring in MPSoC systems, thereby having a higher accuracy of analysis results.

3 Basic Principles

Our main goal is the efficient execution of parallelized applications on a heterogeneous MpSoC architecture. More specifically, the task is to generate an optimal mapping of a parallelized application onto a MpSoC architecture in an automated manner. The most challenging question related to this task is how to obtain the data on which the optimization is based and how to evaluate the performance of specific mappings. We regard traditional HW/SW cosimulation, be it at register-transfer level, cycle accurate, or instruction accurate level, as too slow to be included in an iterative performance optimization process. The integration of fast performance evaluation methods not relying on simulation into the design flow is the biggest challenge in realizing our goal. Leveraging an appropriate application model is the key to achieve our goal. In particular, we have identified the following requirements of the applications and the way they are developed:

- *Parallelization:* The automatic parallelization of sequential program code is still an open research topic and it will not be addressed in this paper. Therefore, the application programmer is required to manually parallelize the algorithm based on the knowledge about the algorithm.
- *Software Refinement and Retargetability:* Developing, testing, and maintaining program code is a major effort in embedded systems design. It is crucial that the platform-independent code basis can be reused without any modification when the hardware platform or single resources are changed. Usually, the platform-dependent code is generated automatically. This requires the application code to be amenable for refinement in different environments and tool-chains.
- *Scalability:* Often, applications are available in different sizes or configurations. Scaling and reconfiguration of an application needs to be supported in a simple way.
- *Mapping:* Usually, there are many ways to map an application onto a given hardware architecture. Mapping an application in different ways should not require any change of the application code but only of the platform-dependent code.
- *Performance Analysis:* Obtaining performance data at the highest possible level of abstraction is one basis for fast mapping optimization. Applications should be written such that certain performance data can be already obtained at application level without the need to change the code.

The Distributed Operation Layer (DOL) can be considered as a framework tailored towards this requirements and that actually implements the mapping optimization. It consists of basically four parts, as shown in Fig. 1. First, the DOL defines how to specify and program applications which is the basis for the automation of the software flow. Applications written according to this definition are amenable for automated refinement with respect to the hardware and the operating system. Second, the DOL defines how to specify the (abstracted) MpSoC hardware architecture. Third, a purely functional simulation can be automatically generated based on the application specification. This simulation can be used for debugging and testing as well as for obtaining mapping relevant parameters at the application level. Fourth, the DOL implements a tool for mapping optimization which itself relies on a tightly coupled tool for performance analysis. Performance analysis sometimes needs to rely on simulation, for instance, to obtain the

actual execution time of code segments. Besides the high-level functional simulation, an external low-level simulation framework is therefore loosely coupled with the mapping optimization tool.
The output of the DOL is a mapping specification which is used together with the application and architecture description as input for the further software refinement.

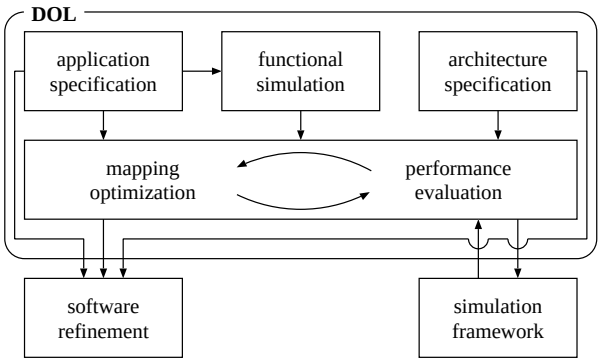

Figure 1. Overview over the DOL framework.

4 Application, Platform and Mapping

4.1 Programming Model and Application Specification

The requirements concerning the programming model have already been listed in the previous section. It turns out that the *process network model* of computation fulfills the best our specification needs.

In terms of *parallelization*, the process network model clearly separates two layers of abstraction, namely inter-process concurrency which can be used for distribution, parallelization and pipelining as well as intra-process parallelism which can be exploited using thread-based processing or instruction-level parallelism.

Concerning *software refinement and retargetability*, tasks can be written in conventional C/C++ and we can rely on state-of-the-art compiler technology for the conversion of program code into machine code. In addition, the process network strictly separates computation from communication, such that the creation of code for a new mapping merely requires a new implementation of the communication routines.

Scalability is supported because of the inherent hierarchical nature of process networks (see above) and the use of iterated processes and communication links.

In terms of *mapping*, there is a clear separation between the structure of the application (defined by the structure of the process network) and the behavior of the application (defined by the code of the processes). This separation considerably simplifies the mapping compared to monolithic applications whose structure is often defined implicitly.

Concerning *performance analysis*, the separation of computation and communication as well as the separation of structure from behavior yields an application that can be broken down into blocks of manageable complexity. These blocks can be analyzed in a structured way, for instance using hierarchical, component based analysis methods.

Structural Specification. The structure of an application is defined by the structure of the process network. With respect to the structure, a process network is a directed graph whose nodes represent the processes and whose directed edges represent the communication channels between the processes. In the DOL, process networks are syntactically represented in an XML format. The main elements are `process` elements, `software channel` elements, and `connection` elements, as shown in Fig. 2. The concept of a process and software channel is not restricted to sequential programs running on computation resources and FIFO-ordered queues. Instead, a process could also be a memory that is accessed via specific software channels to enable random access.

To enable the definition of scalable structures, we use an `iterator` element which is comparable to the `generate` statement in VHDL. The iterator element allows to replicate single elements, but also entire structures. In the example in Fig. 2, for instance, an iterator element is used to create the four connections between the producer and the consumer process.

The structural XML specification also contains application specific information, such as the minimum required size of the channels, and can be annotated with additional data, such as the runtimes of the processes.

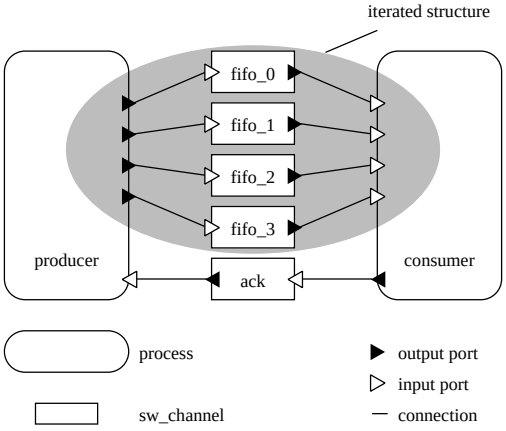

Figure 2. A simple process network with iterated software channels.

Functional Specification. The functionality of an application is defined by the function of the processes. To program the processes, plain C/C++ is used whereby a set of coding rules needs to be respected. In particular, a process consists of an `init` and a `fire` procedure. The `init` procedure

is called once during initialization of the application. Afterwards, the `fire` procedure is called repeatedly. For interprocess communication, dedicated communication primitives have been defined that can be used within `init` and `fire`.

Note that the specification allows to define applications according to *any* process network semantics. Specialized process network semantics, such as Kahn process network or (synchronous) data flow, can be obtained by imposing additional restrictions or semantics onto the elements. When all processes in a process network are structured in the way shown in Algorithm 1, the application has a (synchronous) data flow semantics, for instance.

Algorithm 1 Process Model

```
1: procedure INIT(DOLProcess p)                ▷ initialization
2:     initialize local data structures
3: end procedure

4: procedure FIRE(DOLProcess p)                ▷ execution
5:     DOL_read(INPUT, size, buf)              ▷ blocking read
6:     manipulate
7:     DOL_write(OUTPUT, size, buf)            ▷ blocking write
8: end procedure
```

4.2 Platform Specification

The platform specification captures the information about the underlying platform that is relevant for the mapping. Since the DOL is used for mapping optimization at the system level, the platform does not need to be modeled in every detail but an abstracted model can be used. In particular, the DOL platform specification models the architecture *including* the hardware abstraction layer (or hardware dependent software (HdS)).

The platform specification contains three kinds of data: *structural*, *performance*, and *parametric data*. The structure is defined by the platform's resources, such as processors, memories, hardware channels, and their interconnections, as shown in Fig. 3. Note that instead of specifying local connections and allowing any communication enabled by these connections, end-to-end *communication paths* with nodes on the affected resources are used. This way, Networks on Chip (NoC) can faithfully be modeled. In system-wide performance analysis we are more interested in the performance data of end-to-end connections than in the performance data of the single segments involved. In the example platform in Fig. 3, the two processors can communicate over three different paths, for instance: through the system bus, through the FIFO link and through the memory, via the system bus.

Performance data such as the throughput of buses, the delay of a communication path, processor and bus clock frequencies and overheads of the hardware dependent software layer (device drivers, resource sharing mechanisms) are annotated to the corresponding elements. Parametric data such as memory sizes, supported resource sharing methods (such as FIFO, fixed priority, static scheduling or time triggered architecture), and operating system parameters are annotated to the corresponding elements, too. Like the process network specification of the application, the platform specification is kept in an XML format. In a similar way to the application structure, scalability is supported by using *iterators* that can be used to specify large platforms with a regular structure.

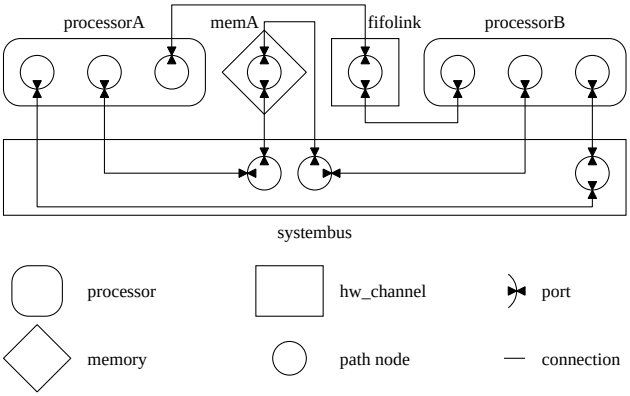

Figure 3. A simple platform specification.

4.3 Mapping Specification

The mapping defines where and how the components of an application are executed on a distributed hardware platform. Mapping needs to be done in the spatial domain, which is referred to as binding, and in the temporal domain, which is referred to as scheduling. In our framework, the binding specification defines a mapping of processes to processors and software channels to communication paths. The scheduling specification defines the scheduling policy on each resource and the according parameters, e.g. time-division multiple access scheme and the associated slot length, fixed priority scheduling and the associated priorities, or static scheduling and the associated ordering.

Again, XML is used as the format for mapping specifications. Besides the possibility to specify iterated mappings, compliant to the algorithm and platform specification, additional mapping parameters, such as the required memory for channels, partitionings of memories, or device driver configuration data, can be specified.

5 Design Space Exploration Cycle

The DOL mapping optimization is an automatic design space exploration process that finally provides an optimized mapping of the application onto the target platform. The iterative design space exploration loop includes two main

phases: performance evaluation (simulation or analysis) and optimization, see Fig. 4.

The decisions during the mapping optimization are based on model data which are are not only collected from system-level simulation or analysis but also from other sorts of platform and application profiling: (a) benchmarking and profiling the hardware platform (e.g. bus peak data rate, communication bandwidth of NoC, available memory), (b) functional simulation of the application software (e.g. the amount of data communicated between processes, the number of process invocations, process dependencies and activation patterns), and (c) the combination thereof (e.g. the run-time of a process on a target processor).

During the design space exploration, the DOL must be able to consider various combinations of (a) processes running on different processors (b) communication links through available communication paths and (c) resource sharing disciplines on these resources. The resulting system performance indicators should be evaluated for each inspected solution. Therefore, there is a need of high-level, fast performance evaluation strategies.

Fig. 4 represents the exploration-estimation cycle. In a first, high-level iteration, the estimation is done using a fast and possibly less accurate analytic performance evaluation method, i.e. an analytic model. This performance evaluation strategy is fast and therefore, it is typically used during early design space exploration.

The second, lower-level iteration utilizes a more detailed simulation in order to collect more accurate performance figures. They can be either used for later stages of the mapping process where a high accuracy is necessary or they can be used to calibrate the parameters of the analytic model, e.g. to determine run-time of processes on computing resources and estimating the scheduling overhead, e.g. context switch costs.

The optimization objectives are a priori fixed by the designer. The DOL optimization can deal with multiple, often conflicting objectives. In addition, it is necessary to take into account constraints on feasible mappings, e.g. in terms of available local memory and fixed locations of some of the processes, e.g. for input/ouput of data. In the current implementation of the DOL, the mapping optimization is based on evolutionary algorithms [27] and the PISA interface [3]. We have considered a two-dimensional optimization space, following two optimization criteria: computation time optimization and communication time optimization, see Sections 6 and 7.

6 Analytic Performance Estimation

In the following, we will describe two possible schemes for evaluating the system-level performance of an implementation, i.e. an application mapped to a hardware platform.

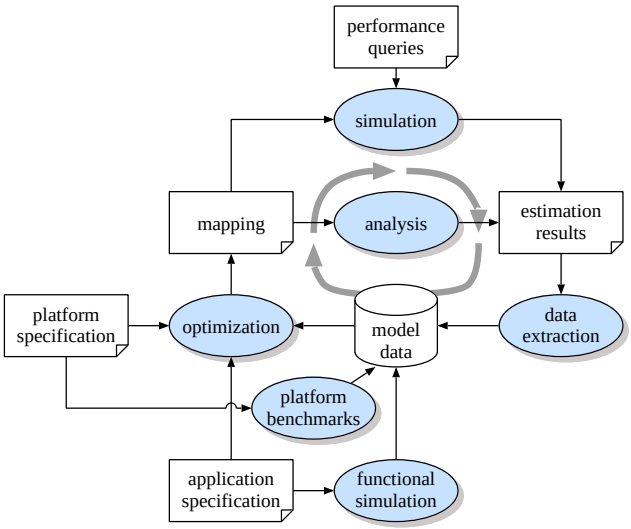

Figure 4. Exploration-estimation cycle.

6.1 Static Model

The first model is static in the sense that it does not take into account (dynamic) resource sharing at all. Therefore, it is close to conventional 'back-of-the-envelope' methods. On the other hand, the conditions under which the results are of reasonable accuracy match typical streaming applications if specified using a process network:

- There is enough inter- and intra-process parallelism available such that the resources do not stall because of blocking, e.g. by missing input data or waiting for memory transfer.
- Overhead for context switching (as for latency hiding) and communication set-up is negligible.
- Communication is smooth, i.e. bursts can be abstracted away for the overall system behavior.

The basic concept of the model is to balance the computation as well as the communication load, i.e. there are two distinct optimization goals. Load balancing avoids overloading certain resources which would lead to excessive packet delays in terms of the communication paths and excessive processing delays at computation resources [25]. Under the above mentioned assumptions, one can also neglect the influence of resource sharing methods as the influence on delay gets significant only when working at the resource limits.

Before determining the objectives for each mapping, some basic quantities need to be determined:

- The whole application is executed using the functional simulation, see Fig. 4, for a set of typical input data. The resulting quantities are for each process $p \in \mathcal{P}$ the number of invocations of the fire-method $n(p)$ and the total

amount of Bytes $b(s)$ communicated on each software channel $s \in \mathcal{S}$.

- Using the simulation component in the exploration cycle in Fig. 4, the runtime $r(p, c)$ of the fire-method of each process $p \in \mathcal{P}$ on each allocatable computation resource $c \in \mathcal{C}$ is determined, using the input traces determined during the functional simulation.
- Using the platform benchmarking, we also determine the maximal throughput $t(g)$ for each segment $g \in \mathcal{G}$ of the communication paths in the hardware platform.

Given a mapping, the two objectives (to be minimized) can be determined.

$$obj_1 = \max_{c \in \mathcal{C}} \left\{ \sum_{\forall p \text{ mapped to } c} n(p) \cdot r(p, c) \right\} \quad (1)$$

$$obj_2 = \max_{g \in \mathcal{G}} \left\{ \sum_{\forall s \text{ mapped onto } g} \frac{b(s)}{t(g)} \right\} \quad (2)$$

Here, obj_1 or obj_2 estimate the runtime if constrained by the maximally loaded computation resource or segment of a communication path, respectively. In the multi-objective evolutionary optimization, we not only consider these two objectives but also additional constraints such as dedicated mappings of processes to processors and delay-sensitive software channels to appropriate hardware channels.

6.2 Dynamic Modular Analysis

The above analysis method is static in the sense that it neglects resource sharing schemes and supposes smooth operation and communication patterns, i.e. without bursts, blocking and synchronization. Recently, there have been major breakthroughs in terms of analysis frameworks that (a) allow system-level performance analysis of distributed embedded systems, (b) are modular in terms of algorithms and hardware components and (c) allow for worst-case analysis of timing behavior. As a prominent example, let us describe the Modular Performance Analysis (MPA), see e.g. [24], [5], that is based on Real-Time Calculus [22], in more detail.

In contrast to other methods, not only properties of traffic streams (flowing through the software/hardware channels) are described in detail but also the capabilities of the used resources. The notion of arrival and service curves captures not only periodic behavior but all kinds of burstiness as well. In particular, for each internal and external stream we define as $R(s, t)$ the number of events in time interval $[s, t)$, $t > s$ and the associated tuple of arrival curves $\alpha = [\alpha^l, \alpha^u]$

$$\alpha^l(t - s) \leq R(s, t) \leq \alpha^u(t - s) \quad (3)$$

In a similar way, one can define the available resource units (e.g. in terms of processing cycles or communicated Bytes) in $[s, t)$, $t > s$ as $C(s, t)$ and the associated tuple of service curves $\beta = [\beta^l, \beta^u]$

$$\beta^l(t - s) \leq C(s, t) \leq \beta^u(t - s) \quad (4)$$

Using the exploration-estimation cycle in Fig. 4, one can determine the characterizations of the input streams using the application specification, e.g. by performing a simulation of the environment and determining the arrival curves of the input streams by using the associated traces and applying (3). In a similar, way, the resource capabilities can be determined using the platform specification and the definition in (4).

The next step is to determine a network of performance analysis components given a mapping according to Fig. 4. The principle of this approach can best be described by a simple example. To this end, Fig. 5 contains an example of such a network. In this case, we have 5 distinct resources, namely 4 CPUs and one BUS. In the network, these resources are represented as sources of the corresponding service curves β. We have one input stream `IN` that is represented by its arrival curve α. In this simplified MPEG-decoder example, the input stream is partitioned into three streams at CPU_1 in process VI which are communicated to CPU_3 ... CPU_4 via a TDMA-scheduled bus. After processing , these streams are communicated back via the bus and combined in CPU_1 using task VO.

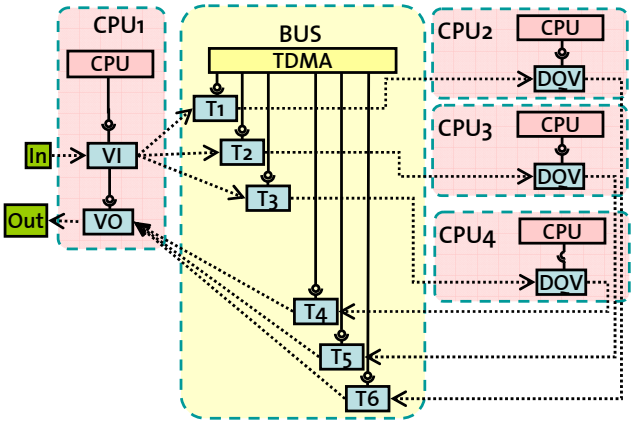

Figure 5. Example of a Modular Performance Analysis network.

The network follows this structure: besides the resources, it contains components that split the available service among the tasks (see e.g. the block TDMA) and performance components (see e.g. VI, VO, T1 ... T6, DOV) that model the use of resources by computation and communication processes. The computations on the arrival curves (flowing horizontally in Fig. 5) and service curves (flowing vertically in Fig. 5) are done using Real-Time Calculus, see

e.g. [24], [5] and [22]. These computations need input from the simulations of processes on the individual resources, see also Section 6.1. This input is needed to relate the events to be communicated or processed and the associated use of resources, e.g. in terms of workload curves.

Based on such a network, one can now determine important performance and memory figures such as end-to-end delays, maximal utilization and the maximal backlog of events. In contrast to the method described in Section 6.1, the MPA allows a much higher accuracy.

7 Case Study — Implementation of an MPEG-2 Decoder

To demonstrate the capabilities of the DOL, we present the implementation of an MPEG-2 video decoder and describe how this decoder is mapped onto a heterogeneous architecture using the static model for performance evaluation.

7.1 MPEG-2 Implementation

Concerning the implementation of the decoder, the main question is how to parallelize the decoding algorithm. The process network in Fig. 6 shows how we exploit the available data parallelism and functional parallelism in the MPEG-2 decoder. Variable length decoding (VLD), inverse scan (IS) and inverse quantization (IQ) are performed on the macroblock level. The inverse discrete cosine transform (IDCT) is performed on the block level, and motion compensation (MC) is again performed on the macroblock level. Additionally, groups of pictures (GOP) can be decoded in parallel.

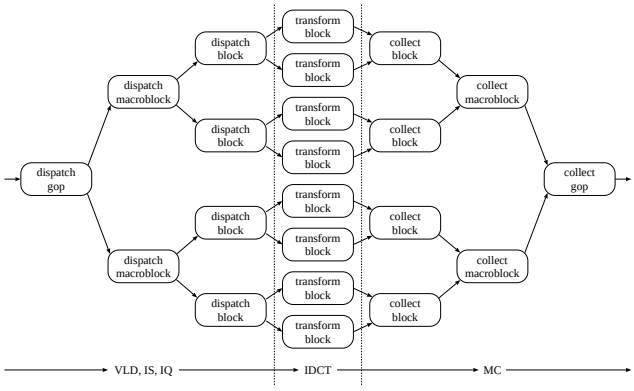

Figure 6. Kahn process network implementation of an MPEG-2 video decoder.

The process network has been implemented as a Kahn process network. It uses only blocking read and write semantics. Bounding the FIFO sizes has been simple since the size of the communicated data structures is known. Also, deadlocks due to the boundedness of the FIFOs can be easily avoided because the process network has a feed-forward structure without cycles. Note that the communication between any two processes is handled via a single FIFO and no global control process or picture buffer exists, leading to a highly parallelized implementation. The process network itself has been defined using *iterators*. Different structures of the process network can be obtained by merely changing the three parameters that define how many outputs each *dispatch gop*, *dispatch macroblock*, and *dispatch block* process has.

7.2 Mapping Optimization

We map the MPEG-2 decoder in the configuration shown in Fig. 8 onto the multiprocessor architecture shown in Fig. 7. The processors in the depicted architecture have a clock frequency of 400 *MHz*, the throughput of the buses is 150 *MByte/s*, and the throughput of the NoC is 100 *MByte/s*. The mapping is computed for MPEG streams with a bit rate of 8 Mbps. The video resolution is 704×576 pixels, and the frame rate is 25 frames per second.

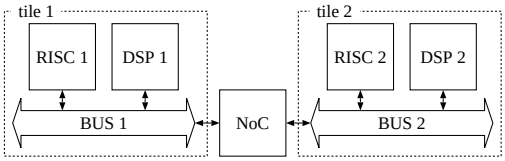

Figure 7. Target architecture in which two identical tiles consisting of a RISC processor and a DSP are connected by a NoC.

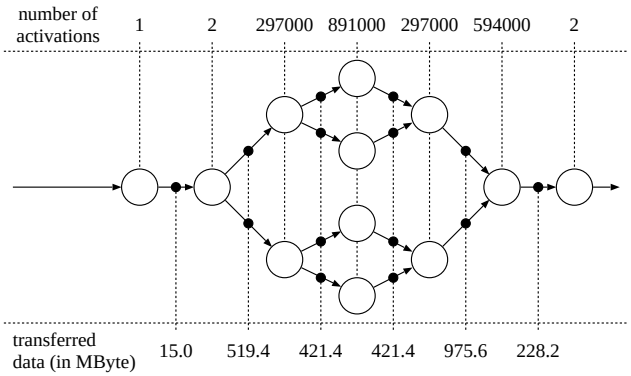

Figure 8. Parameters relevant to mapping as obtained using the functional simulation.

The quantities for the static model, as described in Section 6.1, are summarized in Fig. 8 and Table 1. The number of process invocations $n(p)$ and the total amount of bytes communicated over each software channel are shown in Fig. 8. The runtimes of the processes are listed in Table 1.

Process	Runtime on RISC	Runtime on DSP
dispatch gop	0.13	0.20
dispatch macroblock	6.68	8.52
dispatch block	0.06	0.04
transform block	2.00	1.25
collect block	0.05	0.04
collect macroblock	12.33	8.51
collect gop	0.18	0.30

Table 1. Accumulated runtimes $n(p) \cdot r(p,c)$ of processes for processing a video clip with a duration of 15s.

For the mapping optimization, evolutionary algorithms are used, as already mentioned. Evolutionary algorithms try to find solutions to a given problem by iterating three steps: (a) the evaluation of candidate solutions, (b) the selection of promising candidates based on this evaluation, and (c) the generation of new candidates by variation of these selected candidates. We use the frameworks of EXPO [21] and PISA [3] that implement the iteration shown in Fig. 4 and use the Strength Pareto Evolutionary Algorithm (SPEA) [26] as the underlying multi-objective search algorithm. Using that approach, one only needs to implement parts (a) and (c) of the mapping optimization as (b) is independent of the actual optimization problem and is handled inside SPEA. In the case of EXPO, this is done in three Java classes:

- *Specification*: In the *Specification* class, the static components of the problem are modeled, such as the process network or the architecture.
- *Gene*: In the *Gene* class, a single candidate mapping is represented. In addition the variation methods *crossover* and *mutate* are implemented in this class.
- *Analyzer*: In the *Analyzer* class, the objective values obj_1 and obj_2, see (1, 2), of a single candidate mapping are computed.

Note that these three classes are generated automatically based on the application specification, the architecture specification, and the mapping parameters described above.

7.3 Results

Fig. 9 and Fig. 10 show the result of the mapping optimization. The plots in Fig. 9 show mappings arranged in two dimensions according to their objective values obj_1 and obj_2. Implementations that are located in the shaded region can decode in real-time the $15s$ video used to obtain the performance data. Since a video with a typical workload has been chosen, we can conclude that *any* video with the same parameters could be decoded by these implementations.

Finally, Fig. 10 shows two concrete implementations that achieve the best balance with respect to the computation load (left) and the communication load (right) in the feasible region. Both implementations, however, might be a bad choice for an actual implementation since small variations of the workload can move them out of the feasible region. In contrast, the mapping corresponding to point C in Fig. 9 might be a good choice because by minimizing $\min(obj_1, obj_2)$ it is the mapping that is most robust to workload variations.

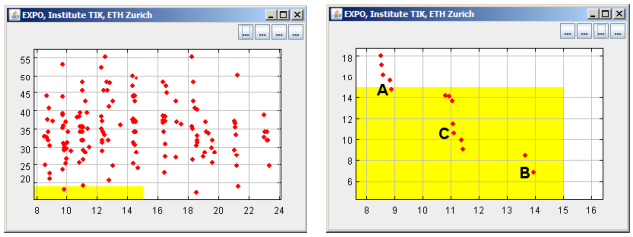

Figure 9. Initial population (left) and optimal mappings (right) obtained after 40 rounds. The x-axis corresponds to obj_1 and the y-axis to obj_2.

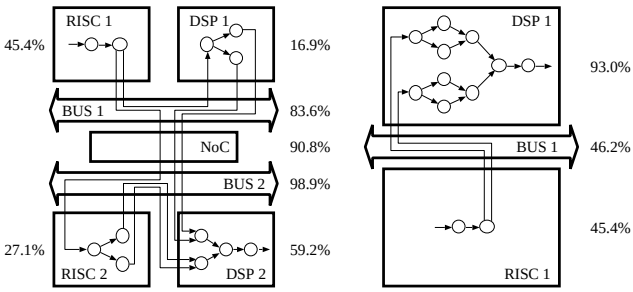

Figure 10. Two optimal mappings corresponding to point A (left) and point B (right). Each resource is annotated with its load, i.e. the ratio between the time it is busy and the totally available time (15s).

8 Conclusions

In this paper, we introduced the Distributed Operation Layer as a framework for specifying and mapping parallel applications onto heterogeneous multiprocessor platforms. We showed how we specify applications based on the process network model of computation and the platform. A component-based approach is used in both specifications which enables the optimization and analysis of mappings by means of analytic performance evaluation. A static model and a dynamic model for analytic performance evaluation have been considered in detail. Finally, we presented a case study in which we used the Distributed Operation Layer for mapping an MPEG-2 video decoder onto a distributed architecture based on the static model.

9 Acknowledgement

The work described in this paper was part of the SHAPES European Project (FET-FP6-2004-IST-4.2.3.4(viii) - started Jan 2006). See www.shapes-p.org for a complete documentation.

References

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[2] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36:45–52, April 2003.

[3] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA – A Platform and Programming Language Independent Interface for Search Algorithms. In C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, volume 2632/2003 of *LNCS*, pages 494–508. Springer-Verlag Heidelberg, 2003.

[4] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2007-7, EECS Department, University of California, Berkeley, January 2007.

[5] S. Chakraborty, S. Künzli, and L. Thiele. A General Framework for Analyzing System Properties in Platform-Based Embedded System Design. In *Proc. 6th Design, Automation and Test in Europe (DATE)*, pages 190–195, Munich, Germany, Mar. 2003.

[6] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded systems. In *Proceedings of the 5th ACM Int'l Conf. on Embedded Software (EMSOFT'05)*, pages 264–272, New York, NY, USA, 2005. ACM Press.

[7] M. Gonzlez Harbour, J. J. Gutirrez Garca, J. C. Palencia Gutirrez, and J. M. Drake Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. In *Proc. 13th Euromicro Conference on Real-Time Systems*, pages 125–134, Delft, The Netherlands, June 2001.

[8] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[9] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis — The SymTA/S Approach. *IEE Proceedings Computers and Digital Techniques*, 152(2):148–166, Mar. 2005.

[11] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP Congress*, North Holland Publishing Co, 1974.

[12] S. Künzli, F. Poletti, L. Benini, and L. Thiele. Combining Simulation and Formal Methods for System-Level Performa nce Analysis. In *Proc. Design, Automation and Test in Europe (DATE'06)*, March 2006.

[13] K. Lahiri, A. Raghunathan, and S. Dey. System-Level Performance Analysis for Designing On-Chip Communication Architectures. *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, 20(6):768–783, June 2001.

[14] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245, Jan. 1987.

[15] E. A. Lee and S. Neuendorffer. MoML: A Modeling Markup Language in XML. Version 0.4. Technical Report UCB/ERL M00/12, University of California at Berkeley, March 2000.

[16] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[17] M. Loghi, F. Angiolini, D. Bertozzi, L. B. nini, and R. Zafalon. Analyzing On-Chip Communication in a MP-SoC Environment. In *Proc. Design, Automation and Test in Europe (DATE'04)*, pages 752–757. IEEE Computer Society, 2004.

[18] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and E. F. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *Computer*, 34(11):57–63, 2001.

[19] Seamless Hardware/Software Co-Verification, Mentor Graphics. `http://www.mentor.com/seamless/`.

[20] The Open SystemC Initiative (OSCI). `http://www.systemc.org`.

[21] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. A Framework for Evaluating Design Tradeoffs in Packet Processing Architectures. In *Proc. 39th Design Automation Conference (DAC 2002)*, pages 880–885, New Orleans, LA, USA, June 2002.

[22] L. Thiele, S. Chakraborty, and M. Naedele. Real-Time Calculus for Scheduling Hard Real-Time Systems. In *Proc. Int'l Symp. Circuits and Systems (ISCAS'00)*, volume 4, pages 101–104, Geneva, Switzerland, Mar. 2000.

[23] The Cadence Virtual Component Co-design (VCC). `http://www.cadence.com/products/vcc.html`.

[24] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System Architecture Evaluation Using Modular Performance Analysis: A Case Study. *Int'l Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, Nov. 2006.

[25] J. Watts and S. Taylor. A Practical Approach to Dynamic Load Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 9(3):235–248, Mar. 1998.

[26] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In *Evolutionary Methods for Design, Optimisation, and Control*, pages 95–100. CIMNE, Barcelona, Spain, 2002.

[27] E. Zitzler and L. Thiele. Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *IEEE Trans. Trans. Evol. Comput.*, 3(4):257–271, Nov. 1999.