Embedding Formal Performance Analysis into the Design Cycle of MPSoCs for Real-Time Streaming Applications

KAI HUANG, WOLFGANG HAID, IULIANA BACIVAROV, MATTHIAS KELLER, and LOTHAR THIELE, ETH Zurich

Modern real-time streaming applications are increasingly implemented on multiprocessor systems-on-chip (MPSoC). The implementation, as well as the verification of real-time applications executing on MPSoCs, are difficult tasks, however. A major challenge is the performance analysis of MPSoCs, which is required for early design space exploration and final system verification. Simulation-based methods are not well-suited for this purpose, due to long runtimes and non-exhaustive corner-case coverage. To overcome these limitations, formal performance analysis methods that provide guarantees for meeting real-time constraints have been developed. Embedding formal performance analysis into the MPSoC design cycle requires the generation of a faithful analysis model and its calibration with the system-specific parameters. In this article, a design flow that automates these steps is presented. In particular, we integrate modular performance analysis (MPA) into the distributed operation layer (DOL) MPSoC programming environment. The result is an MPSoC software design flow that allows for automatically generating the system implementation, together with an analysis model for system verification.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems—*MPSoC*; C.4 [**Performance of Systems**]: Measurement Techniques; D.2.8 [**Metrics**]: Performance Measures

General Terms: Performance, Design

Additional Key Words and Phrases: Multiprocessors, modular performance analysis, performance analysis, design automation

ACM Reference Format:
Huang, K., Haid, W., Bacivarov, I., Keller, M., and Thiele, L. 2012. Embedding formal performance analysis into the design cycle of MPSoCs for real-time streaming applications. ACM Trans. Embedd. Comput. Syst. 11, 1, Article 8 (March 2012), 23 pages.
DOI = 10.1145/2146417.2146425 http://doi.acm.org/10.1145/2146417.2146425

1. INTRODUCTION

To handle the ever-increasing requirements of real-time streaming and signal-processing applications, embedded systems are shifting from uniprocessor designs towards multiprocessor systems-on-chip (MPSoC). While offering high-scale integration, high computing power, and low power consumption, MPSoCs are characterized by their complexity and a large design space. Therefore, performance analysis is required during the entire design trajectory to supply the system designer with the quantitative data underlying the design decisions. In this context, formal performance analysis methods are particularly useful. In contrast to simulation, which is presumably the most widely used method for performance analysis, formal methods enable fast estimations,

Authors' addresses: K. Huang, W. Haid, I. Bacivarov, M. Keller, and L. Thiele, Computer Engineering Group, ETH Zurich, Switzerland, CH-8092; email: haidw@tik.ee.ethz.ch.

DOI 10.1145/2146417.2146425 http://doi.acm.org/10.1145/2146417.2146425

provide coverage of all corner-cases, and allow for giving real-time guarantees. Examples of such formal performance analysis frameworks are modular performance analysis (MPA) [Wandeler et al. 2006], symbolic timing analysis for systems (SymTA/S) [Henia et al. 2005], and holistic scheduling analysis MAST [González Harbour et al. 2001].

A key factor for the successful application of formal performance analysis is *automation*, that is, requiring the appropriate tool support for model generation, calibration, and evaluation. Automation can only be achieved if performance analysis is treated as a "first-class citizen" during the entire design flow. The contribution of this article is a design flow realizing this idea. In particular, MPA is integrated into the distributed operation layer (DOL) [Thiele et al. 2007], which is an MPSoC programming environment based on dataflow process networks [Lee and Parks 1995] targeted at real-time streaming applications. This integration is done in two steps. First, based on the same system specification as used for system software synthesis, an MPA model is generated and calibrated. This requires knowledge about system modeling, as well as tool support for obtaining the required parameters. Second, the analysis model is evaluated to obtain the performance metrics of interest, which requires a concrete implementation of the analysis model. In our design flow, we use the freely available MPA Matlab toolbox [Wandeler and Thiele 2006c] for that purpose. Because an MPA model of a typical MPSoC can be evaluated within the range of seconds, the proposed approach can be used to analyze a single system, as well as multiple systems, during design space exploration. A prototype implementation of the design flow is used to demonstrate the viability of the approach for several applications, including a video (motion JPEG) and an audio application (wave field synthesis).

This article is based on the work described in Haid et al. [2009], in which synchronous dataflow [Lee and Messerschmitt 1987] was considered in which the number of tokens consumed and produced in each activation (firing) of an actor is constant. By using so-called consumption and production curves, this work is extended to dataflow process networks in which the number of tokens consumed and produced in each activation of an actor may vary due to data or state dependencies. Therefore, being able to model this variability allows for the application at the proposed approach for a larger class of applications. Also, so-called workload curves [Maxiaguine et al. 2004] are used to characterize the execution demand of actors, rather than using simple best-case/worst-case execution times. Due to this tighter characterization, over-approximations are reduced, and tighter results can be obtained. Furthermore, a different MPSoC platform is used in this article, and the overhead of the real-time operating system is taken into consideration, demonstrating the generality of the proposed design flow. In addition, this article gives more details on the technical aspects and contains a richer set of experiments.

The remainder of this article is structured as follows. First, related work is reviewed in Section 2, and the background of our work is presented in Sections 3 and 4 by describing the DOL design flow and the MPA framework, respectively. Section 5 describes the class of considered MPSoCs and how they can be modeled in MPA. In Sections 6 and 7, the automated generation and calibration of formal performance analysis models for this class of MPSoCs is presented. Finally, the viability of the approach is demonstrated with several applications in Section 8, and conclusions are drawn in in Section 9.

2. RELATED WORK

Many model-based frameworks have been developed for the design of multiprocessor embedded systems; see Densmore et al. [2006] and Gerstlauer et al. [2009] for comprehensive surveys. One can observe that the most widely used approach for system performance analysis in these frameworks is simulation-based methods. Unfortunately, simulation suffers from long runtimes and high setup efforts for each new architecture and mapping. Worst-case bounds of system properties, such as 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. Focusing on model-based frameworks based on dataflow process networks, the two frameworks most closely related to DOL are Artemis [Pimentel et al. 2006] and Koski [Kangas et al. 2006]. In these frameworks, performance estimation is also done by simulation at different levels of abstraction.

To achieve shorter runtimes for simulation-based methods, approaches that combine simulation and analysis have been proposed. Lahiri et al. [2001] proposed a hybrid trace-based simulation methodology for on-chip communication exploration. Künzli et al. [2006] developed a technique that replaces single subsystems in a simulator with a corresponding MPA model. Although these mixed methodologies can help shorten the runtime of simulations, the problem of insufficient corner-case coverage still remains.

Instead of simulation, we use a formal method for performance analysis whose integration into a complete design cycle has received much less attention. So far, work in this direction has been mainly done in the domain of (best-effort) networking systems. In that domain, layered queuing networks [Petriu et al. 2000], Petri nets [Woodside 2007], or state-based formalisms [Viehl et al. 2006] have been derived from synthesizable specifications, such as the ITU-T specification and definition language (SDL). Balsamo et al. [2004] provide a survey, illustrating how these methods are integrated into the software development process. In the domain of heterogeneous multiprocessor systems, the UML-MAST tool can generate a model for holistic scheduling based on a system model described in the UML profile for schedulability, performance, and time (SPT) [González Harbour et al. 2001]. However, since UML-SPT models are, in general, not synthesizable, the question of how to automatically obtain the required model parameters remains open.

The second central aspect of this article is model calibration, which is a well-known technique in many modeling and simulation domains. In the context of MPSoC design and analysis, calibration has been applied for low-level models, such as cycle-accurate simulation [Black and Shen 1998], as well as for high-level models, such as trace-based simulation [Pimentel et al. 2008] (in the Artemis framework) or task graphs [Kangas et al. 2006] (in the Koski framework), for instance. The calibration of formal performance analysis models has only been addressed to a very limited extent. The reason being that formal performance analysis has been mainly applied during early design space exploration, where estimation, rather than calibration, is used to determine model parameters, because an actual implementation might not be available at early design stages.

3. DOL PROGRAMMING ENVIRONMENT

For implementing parallel applications on an MPSoC platform, the distributed operation layer (DOL) programming environment [Thiele et al. 2007] is used. The DOL is a platform-independent MPSoC programming environment targeted at real-time streaming and (array) signal-processing applications. It is based on the dataflow process network model of computation [Lee and Parks 1995] and provides source-to-source code generators to efficiently execute DOL applications on different MPSoC platforms. Concretely, the supported platforms are the MPARM platform [Benini et al. 2005], the Sony/Toshiba/IBM Cell BE [Kahle et al. 2005], and the Atmel DIOPSIS 940 [Paolucci et al. 2006].

Following the X-chart paradigm [Gerstlauer et al. 2009], the DOL environment not only distinguishes between the system application and architecture specifications but also between the system implementation and the corresponding performance analysis model, which are both derived from the same specification. Figure 1 gives an overview of the DOL environment. The design cycle starts with a system specification consisting

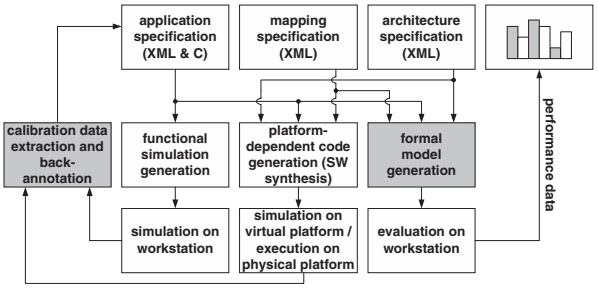

Fig. 1. Overview of the DOL programming environment. The contributions of this article are indicated by the shaded boxes: generation and calibration of formal performance analysis models.

of an application, an architecture, and a mapping specification. The application specification based on the dataflow process network model of computation is platform-independent and needs to be related to a concrete architecture by explicit mapping. Further details on the DOL specification are given later in this section.

The second step in the design flow is the automated generation of a functional simulation of the application. The functional simulation allows for testing and debuging the parallel application code with standard debugging tools on a standard workstation. Specifically, we use the *SystemC* simulation engine and *gdb*.

Once the application is functionally correct, it can be mapped onto the target platform. Based on the architecture and the mapping specification, software synthesis generates the corresponding binaries. This basically involves the generation of mapping-dependent source code for the processors, compilation, and the linking to the platform-specific libraries and runtime environment. The generated binaries can either be executed on a simulator of the target platform (available for MPARM, Cell BE, and DIOPSIS 940) or on the real MPSoC (Cell BE and DIOPSIS 940).

The DOL design flow described so far is similar to related design flows, such as Artemis [Pimentel et al. 2006] or Koski [Kangas et al. 2006]. The unique feature of the DOL design flow is the embedding of formal performance analysis, namely MPA, into the design cycle, as shown in Figure 1. The generation and calibration of MPA models will be explained in detail in Sections 6 and 7. Because applications can be typically mapped onto MPSoCs in different ways, the DOL provides tools that support the developer in exploring the performance of several mappings and choosing the most suited one. For detailed explanations about the used exploration techniques, we refer to Thiele et al. [2007]. Finally, the DOL programming environment features a graphical frontend enabling an easy system specification. Figure 2(a) displays a snapshot of the graphical environment in which a simple parallel application consisting of three processes connected via communication channels is mapped to MPARM.

The rest of this section presents some details of the DOL programming model for applications, the specification of architectures, and the mapping of applications to architectures.

3.1. Application Programming Model

To model streaming applications, the dataflow process network model of computation [Lee and Parks 1995], a subclass of Kahn process networks [Kahn 1974], has been adopted in the DOL design flow. The dataflow model assumes a network of concurrent and autonomous actors communicating in a point-to-point fashion via unbounded first in, first out channels.

Using the dataflow model for programming MPSoCs is beneficial from the perspectives of software development, synthesis, and analysis. With respect to software

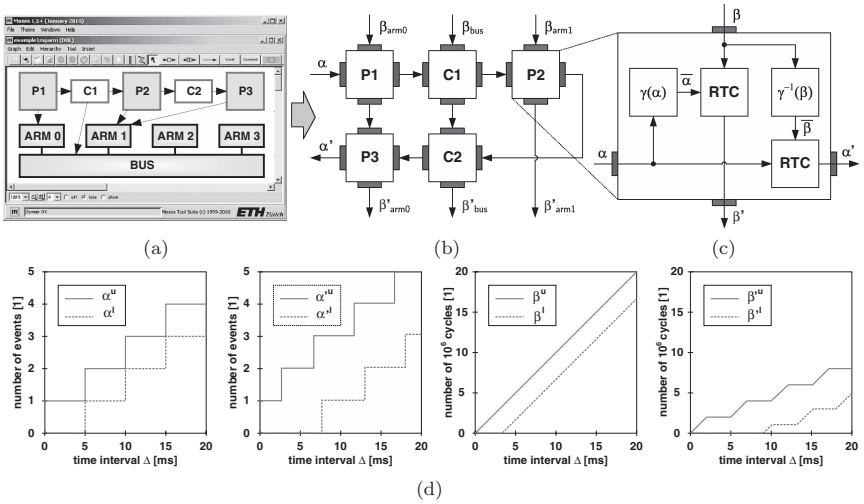

Fig. 2. (a) Screenshot of the graphical frontend of DOL showing a simple dataflow process network consisting of three actors mapped to MPARM. (b) (Automatically generated) MPA model of this system where preemptive fixed-priority scheduling is used on all resources. (c) Greedy processing component for modeling the actor P2 in MPA. (d) Input and output arrival and service curves of the greedy processing component modeling P2.

development, the dataflow model directly exposes the available data and functional parallelism in an application, enabling the efficient parallelization of algorithms. Moreover, the dataflow model is in line with established software engineering practices by ensuring compositionality of actors, because every actor completely encapsulates its own state, together with the code that operates on it. This facilitates code reuse and the development of modular, scalable applications. With respect to software synthesis, the essential property of the dataflow model is its determinism, that is, the result of a computation depends only on the provided input and not on the timing of different processes or their communication. Requiring no global synchronization, the only platform-specific elements to implement are first in, first out channels and a mechanism for executing multiple actors on a processor (if multiple actors need to be executed on each processor). Finally, with respect to system analysis, the modularity and the explicit separation of computation and communication enable a modular generation and calibration of the analysis model.

The dataflow model can be seen as a coordination model which allows for the consideration the programming of a parallel system as the combination of two distinct activities: the *computation* comprising a number of processes involved in manipulating data and the *coordination* describing the connections of processes. To specify dataflow applications, DOL uses two distinct languages, namely C/C++ to program actors and XML for describing the topology of the dataflow process network. Examples for both are shown in Listing 1 and Listing 2. The choice of these languages has pragmatic reasons, as using C/C++ allows to reuse of existing legacy code, and XML is easy to handle, due to the large number of available tools. Alternative choices for the application specification would be domain-specific languages, such as Simulink or StreamIt [Thies et al. 2002] or other modeling languages, such as UML.

3.2. Architecture and Mapping Modeling

The modeling of an *architecture* in DOL is done at an abstract level, the granularity of which depends on the refinements that take place during the software synthesis for

Listing 1. The XML source code of the dataflow process network in Figure 2(a).

```
1   <process name="p1">
2       <port type="output" name="out"/>
3       <source type="c" location="p1.c"/>
4   </process>
5
6   <process name="p2">
7       <port type="input" name="in"/>
8       <port type="output" name="out"/>
9       <source type="c" location="p2.c"/>
10  </process>
11
12  <process name="p3">
13      <port type="input" name="in"/>
14      <source type="c" location="p3.c"/>
15  </process>
16
17  <channel name="c1">
18      <source name="p1" port="out"/>
19      <target name="p2" port="in"/>
20  </connection>
21
22  <channel name="c2">
23      <source name="p2" port="out"/>
24      <target name="p3" port="in"/>
25  </connection>
```

Listing 2. C code of the actor P2 in Figure 2(a). For each invocation (FIRE), P2 reads a float from its input, squares it, and sends the result to its output.

```
1   //process handler (constructor)
2   DOLProcess P2 = {
3       &P2_state,
4       P2_init,
5       P2_fire
6   };
7
8   void P2_init(DOLProcess *p) {
9       //nothing to do
10  }
11
12  void P2_fire(DOLProcess *p) {
13      float i;
14
15      DOL_read((void*)PORT_IN, &i,
16          sizeof(float), p);
17      i = i * i;
18      DOL_write((void*)PORT_OUT, &i,
19          sizeof(float), p);
20  }
```

a specific target architecture. The DOL architecture specification consists of basic system components, their attributes, and the way they are connected, that is, computation resources like programmable processors and hardware IPs, or communication components like buses and NoCs. Figure 2(a), for instance, represents a simplified view of the MPARM architecture. This specification gets more complex when further architectural features are considered during software synthesis. An example are several communication possibilities, such as CPU-driven transfers or transfers via a direct memory access (DMA) controller.

The *mapping* describes the binding of processes and software channels to architecture components and the scheduling on shared resources. For example, scheduling policies like time-division multiple access, (non-)preemptive fixed priority, or earliest deadline first and the corresponding parameters, can be specified.

Customized XML schemata are used for describing the format of architecture and mapping specifications. These specifications are used as inputs for both software synthesis and analysis model generation.

4. MODULAR PERFORMANCE ANALYSIS

In the domain of real-time streaming and digital signal-processing applications, powerful abstractions have been developed to model and analyze distributed systems. The framework used in this article is modular performance analysis (MPA) [Wandeler et al. 2006]. With MPA, hard upper and lower bounds can be computed for various performance criteria in a real-time system, such as end-to-end delays, buffer requirements, or resource utilization. Hence, MPA qualifies for the analysis of hard real-time systems. This clearly distinguishes MPA from any probabilistic performance analysis method and from performance estimation by simulations.

Basically, the analysis of real-time systems requires knowledge about the best-case and worst-case behavior of the system, in all operating conditions. In MPA, a compositional approach is employed to tackle this problem. A system is split up into small components, with no or little interference, that are characterized independently

from each other. To characterize the components in terms of best-case and worst-case behavior, data sheets, exhaustive simulation, or formal methods can be used, as will be shown in Section 7. Afterwards, the interference of components is modeled to globally analyze a system. This takes into account the interaction of software and hardware components with respect to data dependencies, operating system overhead, or resource sharing, as described next.

Concretely, the MPA model of a system is composed of abstract elements that model (a) event streams that carry data and trigger actors, (b) the computation and communication of an application, (c) resources such as processors and interconnects, and (d) resource sharing methods. The approach uses real-time calculus (RTC) [Chakraborty et al. 2003], which itself is based on the theoretical framework of network calculus [Le Boudec and Thiran 2001]. In particular, arrival curves $\alpha(\Delta)$, service curves $\beta(\Delta)$, and workload curves $\gamma(\Delta)$ model timing properties of event streams, the capability of architecture elements, and the execution requirements of event streams, respectively, as shown in Figure 2(c). Abstract components define the semantics of task execution and resource sharing in the system. A short description of these elements is given next. For further details, we refer to Wandeler et al. [2006].

4.1. Event Stream Model

Event streams in a system can be described using a cumulative function $R(s,\ t)$, defined as the number of events seen in the time interval $[s,\ t)$. While any R always describes one concrete trace of an event stream, a tuple $\alpha(\Delta) = [\alpha^u(\Delta),\ \alpha^l(\Delta)]$ of upper and lower *arrival curves* provides an abstract event stream model that characterizes a whole class of (nondeterministic) event streams. $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$ provide an upper and a lower bound on the number of events seen on the event stream in any time interval of length Δ:

$$\alpha^l(t-s) \leq R(s,\ t) \leq \alpha^u(t-s) \quad \forall s < t \quad s,\ t \in \mathbb{R}^+, \tag{1}$$

with $\alpha^l(\Delta) = \alpha^u(\Delta) = 0$ for $\Delta \leq 0$. Arrival curves substantially generalize traditional event models, such as sporadic, periodic, periodic with jitter, or any other arrival pattern, with deterministic timing behavior. Therefore, they are suited to represent the complex characteristics of event streams in real MPSoCs. In Figure 2(d), for instance, α represents the arrival curves for a periodic stream, whereas α' represents the arrival curves of a periodic stream with jitter.

4.2. Resource Model

In a similar way, the capability of computation or communication resources can be described by a cumulative function $C(s,\ t)$, defined as the number of available resources, that is, processor or bus cycles, in the time interval $[s,\ t)$. To provide an abstract resource model that models a whole set of possible resource behaviors, a tuple $\beta(\Delta) = [\beta^u(\Delta),\ \beta^l(\Delta)]$ of upper and lower *service curves* is defined as

$$\beta^l(t-s) \leq C(s,\ t) \leq \beta^u(t-s) \quad \forall s < t \quad s,\ t \in \mathbb{R}^+, \tag{2}$$

with $\beta^l(\Delta) = \beta^u(\Delta) = 0$ for $\Delta \leq 0$. Again, service curves substantially generalize classical resource models, such as the bounded delay or the periodic resource model. In Figure 2(d), β represents the service curves for a bounded delay resource, whereas β' represents the irregular service that would be left for a task with lower priority.

4.3. Workload Model

To relate arrival and service curves, *workload curves* are used [Maxiaguine et al. 2004]. The workload that an actor imposes on a resource can be described by a cumulative function $W(s,\ t)$, defined as the number of clock cycles required to process $t - s$ consecutive events on a computation or communication resource. We define a tuple

$\gamma(e) = [\gamma^u(e),\ \gamma^l(e)]$ of upper and lower workload curves as:

$$\gamma^l(t-s) \le W(s,\ t) \le \gamma^u(t-s) \quad \forall s < t \quad s,\ t \in \mathbb{N}_0. \tag{3}$$

In the context of MPA, the arrival curve is event-based, whereas the service curve is resource-based. Using the workload curve and its pseudoinverse,

$$(\gamma^u)^{-1}(w) = \sup\{e : \gamma^u(e) \le w\} \qquad (\gamma^l)^{-1}(w) = \inf\{e : \gamma^l(e) \ge w\}, \tag{4}$$

Equations (5) and (6) describe how arrival and service curves can be transformed from event-based to resource-based quantities and vice versa, as shown in Figure 2(c).

$$\bar{\alpha}^l(\Delta) = \gamma^l\big(\alpha^l(\Delta)\big) \qquad \bar{\beta}^l(\Delta) = (\gamma^u)^{-1}\big(\beta^l(\Delta)\big), \tag{5}$$

$$\bar{\alpha}^u(\Delta) = \gamma^u\big(\alpha^u(\Delta)\big) \qquad \bar{\beta}^u(\Delta) = (\gamma^l)^{-1}\big(\beta^u(\Delta)\big). \tag{6}$$

In the simplest case, the workload of an actor is characterized by its worst-case (WCET) and best-case execution time (BCET), measured in clock cycles. In this case, the workload curves would simply be

$$\gamma^l(e) = \mathrm{BCET} \cdot e \text{ [cycles]} \qquad (\gamma^l)^{-1}(x) = \lceil x/\mathrm{BCET} \rceil \text{ [events]}, \tag{7}$$

$$\gamma^u(e) = \mathrm{WCET} \cdot e \text{ [cycles]} \qquad (\gamma^u)^{-1}(x) = \lfloor x/\mathrm{WCET} \rfloor \text{ [events]}. \tag{8}$$

4.4. Actor Model

In MPA, actors that are executed on processors are modeled as *greedy processing components* (GPC). The semantics of a GPC can be described as follows. An incoming event stream, represented as an upper and a lower arrival curve, flows into the first in, first out input buffer of the GPC. The events trigger an actor whose execution is restricted by the availability of the resource, represented by an upper and a lower service curve. The output event stream can again be represented as an upper and a lower arrival curve, while the remaining resource capacity can be represented as an upper and a lower service curve. As has been shown in Chakraborty et al. [2003], the output arrival curves α' can be computed by Equations (9) and (10). Similarly, relations for the output service curve β' can be computed as

$$\alpha'^l(\Delta) = \min\Big\{ \inf_{0\le\mu\le\Delta} \Big\{ \sup_{\lambda>0}\{\alpha^l(\mu+\lambda) - \bar{\beta}^u(\lambda)\} + \bar{\beta}^l(\Delta-\mu)\Big\},\ \bar{\beta}^l(\Delta)\Big\}, \tag{9}$$

$$\alpha'^u(\Delta) = \min\Big\{ \sup_{\lambda>0}\Big\{ \inf_{0\le\mu<\lambda+\Delta}\{\alpha^u(\mu) + \bar{\beta}^u(\lambda+\Delta-\mu)\} - \bar{\beta}^l(\lambda)\Big\},\ \bar{\beta}^u(\Delta)\Big\}. \tag{10}$$

Local quantities describing a component's performance can be derived analogously. For example, an upper bound of the maximum delay $d^{\max}$ experienced by an event and the maximum backlog (buffer fill level) $b^{\max}$ at a GPC are given by the following relations; see also Le Boudec and Thiran [2001].

$$d^{\max} = \sup_{\lambda\ge0}\big\{\inf\{\tau \ge 0 :\ \alpha^u(\lambda) \le \beta^l(\lambda+\tau)\}\big\}, \tag{11}$$

$$b^{\max} = \sup_{\lambda\ge0}\{\alpha^u(\lambda) - \beta^l(\lambda)\}. \tag{12}$$

4.5. Performance Analysis

At this point, we know how to model event streams and computation and communication resources, as well as single HW/SW components (actors). In order to analyze the performance of an entire system, an abstract performance model needs to be built. To obtain this model, all event streams are modeled by arrival curves, and all computation and communication resources by service curves, as well as all actors in the system

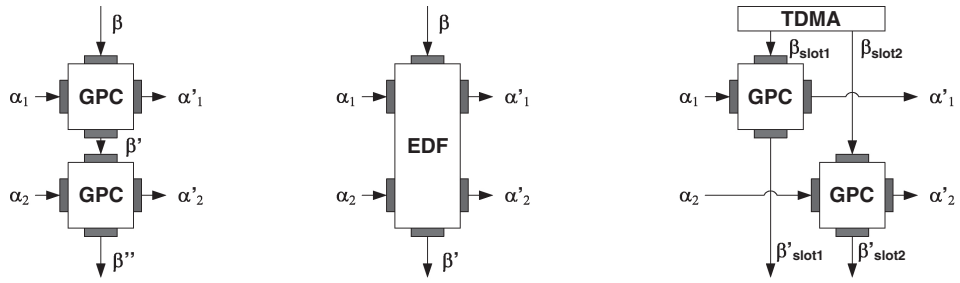

Fig. 3. Modeling of different scheduling policies in MPA. From left to right: preemptive fixed priority, earliest deadline first, and time division multiple access scheduling.

by GPCs. This way, one can analyze distributed systems with any number of shared computation and communication resources. Resource-sharing policies currently supported by MPA include (non-)preemptive fixed-priority scheduling (FP) [Wandeler et al. 2006; Haid and Thiele 2007], rate monotonic scheduling (RM) [Wandeler et al. 2006], time division multiple access (TDMA) [Wandeler and Thiele 2006b], earliest deadline first (EDF) [Wandeler and Thiele 2006a], and first-come first-serve (FCFS) [Perathoner et al. 2010]; see Figure 3.

By correctly interconnecting all arrival and service curves, one obtains the performance analysis model of a system, as shown in Figure 2(b). Based on the local analysis of single components, global system properties (such as end-to-end delays, total buffer requirements, system throughput, and others) can be computed.

Tool support for MPA is available as a Matlab toolbox that implements the basic operations from real-time calculus [Wandeler and Thiele 2006c]. Based on these operations, the Matlab toolbox provides methods for curve generation, implements the analysis for the scheduling policies just mentioned, and provides support for plotting arrival and service curves.

Finally, taking state information into account would sometimes allow for improving the analysis, because safe (but overly pessimistic) assumptions could be relaxed using this knowledge. Hence, first techniques that combine stateful models (finite state machines or timed automata) with compositional methods have been proposed; see Lampka et al. [2010] and the references therein, for instance. Automated generation and calibration of models that are amenable to these techniques are beyond the scope of this article, however.

5. MODELING MPSOCS IN MPA

In the previous section, the basic abstractions and modeling elements of MPA have been reviewed. Concepts like event streams and processing components match well the basic behavior of a distributed system. They are also sufficient for modeling systems during early design space exploration. For accurately modeling real MPSoCs, however, a basic model, such as the one shown in Figure 2(b), is too simplistic, Actors only have a single input and output, token production and consumption rates are constant, and the operating system overhead is neglected.

By extending the MPA concepts introduced in the previous section, this section shows how an MPSoC can be modeled in MPA. It is assumed that all the data for doing so are already available, that is, calibration has already been performed (see Section 7). This section is limited to dataflow process networks modeled in the framework of DOL and executing on MPARM [Benini et al. 2005]. Even using extensions of MPA, however, not all systems that can be modeled in DOL (and implemented on MPARM using DOL's

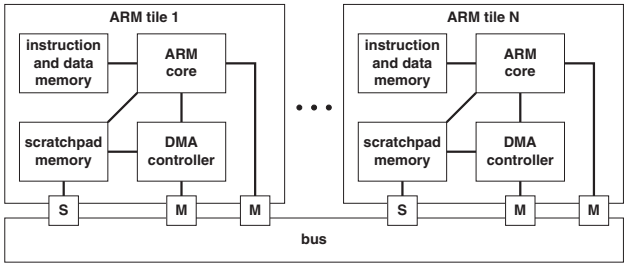

Fig. 4. Block diagram of the considered MPARM architecture. The blocks labeled with "M" and "S," respectively, denote master and slave ports on the bus.

software synthesis backend) and can be analyzed using MPA. Therefore, this section also describes the class of systems that can be currently modeled and analyzed in our implementation of the model generation and calibration framework. Note that due to the compositionality of MPA, incorporating new analysis techniques to extend the scope of systems is possible. Thus, the proposed framework is not limited by the difficulty of integrating new techniques for MPA, but rather by the difficulty of developing these techniques.

5.1. Hardware/Software System

A hardware/software platform amenable to compositional analysis should keep the interference between components to a minimum and allow for a tight characterization of components. Clearly, this should not overly sacrifice system performance or increase the cost of the system. In the following, an MPARM architecture and the according runtime environment for DOL applications that have been designed with these goals in mind are described. MPARM has a distributed memory architecture that is typical for MPSoCs and has been implemented as a cycle-accurate simulator. MPARM consists of a set of fully programmable ARM tiles and a shared interconnect, as shown in Figure 4. The platform can be easily configured in different ways. We used the configuration described in the following to reduce the interference between tiles, between the actors executing on each tile, and between computation and communication.

In MPARM, each tile has its own local program and data memory, such that it can operate independently from the other tiles. The interference between tiles is thus limited to data transfers triggered by the application. RTEMS (real-time executive for multiprocessor systems) [RTEMS Steering Committee 2010] is used as the real-time operating system. The actors of a DOL application that are bound to each tile are implemented as RTEMS tasks. The channels of a DOL application are implemented as RTEMS message queues. In the analysis, the booting of the system is neglected. Afterwards, a steady-state operation is reached, during which RTEMS merely reacts to interruptions caused by communication events and switches between tasks. During steady-state operation, the only processing cost caused by the real-time operating system is thus the context-switching overhead, which is explicitly modeled as shown in Section 5.4. Further activities performed by RTEMS could be modeled as separate tasks in the analysis model.

To reduce the *interference between actors* executing on the same tile, caches are disabled, and processors read instructions and data directly from a local (scratchpad) memory. This is, for instance, also the case for the Sony/Toshiba/IBM Cell BE [Kahle et al. 2005], in which the eight synergistic processing elements do not have a cache. Similarly, the DSP subsystem on the Atmel DIOPSIS 940 does not have a cache either [Paolucci et al. 2006]. This way, the execution time of actors does not depend on the

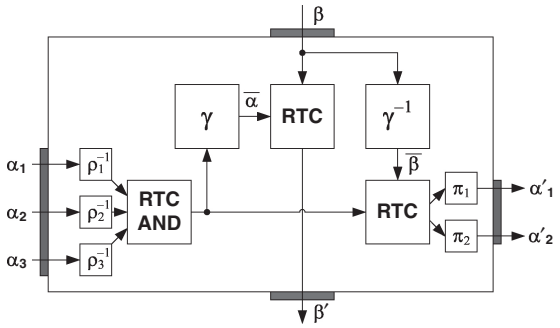

Fig. 5. Multiple-input multiple-output GPC to model the actor shown in Listing 3.

cache, which is influenced by other actors running on the same processor and their scheduling, resulting in tighter bounds of the best-case/worst-case execution time. Of course, using the memory hierarchy by enabling caches usually improves the average case behavior substantially and may also reduce the worst-case runtimes of actors. In this case, the technique presented in Pellizzoni and Caccamo [2007] could be used for the worst-case performance analysis.

To reduce the interference between computation and communication, direct memory access (DMA) controllers are used. Located at the interface between a tile and the interconnect, a DMA controller can autonomously handle the message queue communication between actors without requiring processor resources. In addition, the DMA controller can access the scratchpad memory in parallel and independently of the processor. If the processor and the DMA controller mutually influenced each other, the memory could be modeled as a separate resource.

5.2. Modeling Computation

The GPC introduced in Section 4.4 can be seen as the basic entity to model actors of a dataflow process network. A GPC that accepts a single input event stream and produces a single output stream is not sufficient for tackling dataflow applications, however. In general, actors in a dataflow process network might have more than one input and possibly also more than one output. This is shown in Listing 3, which lists the actor model considered in this article. An actor is fired repeatedly when data is available on all of its input channels. Upon completion of the actor, data is written to all of its output channels. Furthermore, the number of accesses to each input and output channel may vary between firings, which is common for multimedia streaming applications.

Listing 3. Structure of an actor's FIRE function.

```
1 void fire() {
2   DOL_read(input[1], buffer_in[1], N_in[1]);
3   DOL_read(input[2], buffer_in[2], N_in[2]);
4   DOL_read(input[3], buffer_in[3], N_in[3]);
5   execute;
6   DOL_write(output[1], buffer_out[1], N_out[1]);
7   DOL_write(output[2], buffer_out[2], N_out[2]);
8 }
```

The model of a multiple-input multiple-output GPC corresponding to the actor in Listing 3 is illustrated in Figure 5. To model variable access rates, a data consumption

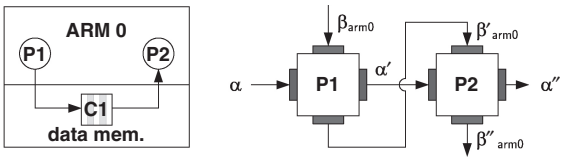

Fig. 6. Intra-processor communication and corresponding MPA model.

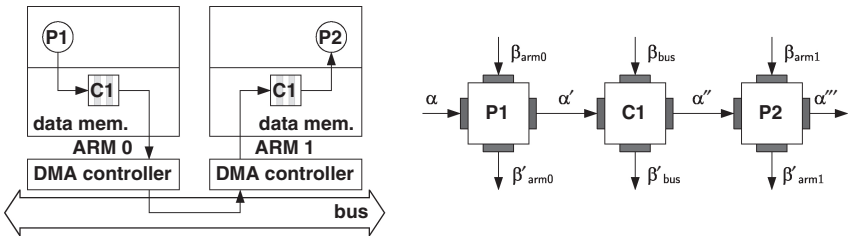

Fig. 7. Inter-processor communication and corresponding MPA model.

curve ρ and a data production curve π are used. The data consumption curve $\rho(e) = [\rho^u(e),\ \rho^l(e)]$, $e \in \mathbb{N}_0$ specifies the maximum and minimum number of read accesses to an input channel for e consecutive activations of the actor. The data production curve $\pi(e) = [\pi^u(e),\ \pi^l(e)]$, $e \in \mathbb{N}_0$ specifies the maximum and minimum number of write accesses to an output channel for e consecutive activations of the actor. Note that while consumption and production curves allow for analyzing dataflow process networks with MPA, the increased generality of the model also incurs a loss of analysis capabilities. As an example, techniques to analyze effects in systems with finite buffers (blocking write, back-pressure) that are limited to synchronous dataflow [Thiele and Stoimenov 2009] cannot be applied any more when token consumption and production rates are not constant.

To join multiple inputs, AND components (see Section 6.2) are used. The analysis of this component has been described in Haid and Thiele [2007]. Since its interface still only consists of arrival and service curves, the analysis of the entire system can be performed in the same way as a network of single-input single-output GPCs.

5.3. Modeling Communication

Besides computation, also the communication between actors also needs to be considered in the generation of a system model. In MPARM, the implementation of communication between actors located on the same tile (*intra-processor communication*) is different from the implementation of communication between actors located on different tiles (*inter-processor communication*). These two cases are considered next.

The left hand side of Figure 6 shows the implementation of intra-processor communication. The channel buffer is allocated in the local memory of the processor. After the producer has written new data to the channel, the consumer can read the data as soon as it is scheduled. This behavior can be modeled by chaining two GPCs, as shown in Figure 6, in which fixed priority scheduling is assumed for illustration purposes.

Inter-processor communication involves four hardware resources, namely the two processors where the producer and the consumer are located, the DMA controller of the processor where the producer is located, and the interconnect, as shown in Figure 7. The communication protocol is implemented in software as follows. The

producer puts data into its local memory and notifies the DMA controller to carry out the transfer. Upon notification, the DMA controller reads the data and transfers them to the local memory of the processor where the consumer is located (in both cases using direct memory access that does not require any processor resources). From its local data memory, the consumer can then read the data. This behavior is modeled by chaining the two GPCs representing the actors with an intermediate GPC located on the interconnect, as shown in Figure 7. Since the DMA controller just acts as the arbiter for the interconnect, it is not modeled explicitly. In contrast, the interconnect itself is a shared resource whose availability is modeled using a dedicated service curve.

5.4. Modeling Context-Switch Overhead

The previous sections showed how actors and their communication are modeled in MPA. It remains to show how the overhead to context switch between actors can be safely considered in the performance analysis. Basically, one can distinguish between event-triggered scheduling policies (FP, EDF, FCFS), in which context switches occur depending on the availability of data, and time-triggered scheduling (TDMA), in which context switches occur on a regular time basis.

First, event-triggered scheduling is considered. In this case, a (safe) optimistic assumption about the best case is that no context switch at all needs to be taken into account for a task execution. In other words, the function FIRE in Listing 3 is continuously repeated and executed without any interference from other tasks.

A (safe) pessimistic assumption about the worst case is that starting, as well as completing a task (i.e., the execution of the function FIRE in Listing 3) results in a context switch. Such a context switch involves the execution time of the call to the underlying operating system, saving the context of the currently running task, and restoring the context of the starting task. Note that the context-switch overhead is only considered in the preempting task and not in the preempted one. As a result, even if an actor is preempted several times, only two context switches are considered in its worst-case execution time.

Context switches can thus be modeled by adding the worst-case context-switch time $T_{context}$ to the upper workload curves of all actors, while the lower workload curves do not need to be modified. Formally, the workload curve $\tilde{\gamma}$, including the overhead for context switches, is thus related to the original workload curve γ by:

$$\tilde{\gamma}^u(e) = \gamma^u(e) + 2 \cdot e \cdot T_{context} \qquad \tilde{\gamma}^l(e) = \gamma^l(e). \tag{13}$$

In TDMA scheduling, context switches can be simply modeled by reducing the length of the timeslots by a lower and an upper bound on the context-switch time [Wandeler and Thiele 2006b]. Contrary to event-triggered scheduling in which the context-switch overhead is modeled in the workload curves of actors, the service curves representing the different slots are modified in TMDA scheduling.

6. ANALYSIS MODEL GENERATION

The seamless integration of performance analysis into the MPSoC design flow requires the generation of performance analysis models. Automating the analysis model generation is highly desirable: On the one hand, models for real systems have a complexity that makes their generation a tedious, error-prone process if done manually. On the other hand, usually different implementation alternatives need to be analyzed for a design, each requiring its own analysis model. This, of course, also applies to automated design space exploration.

Unfortunately, analysis model generation is not a straightforward source-to-source transformation. First, there is not a simple one-to-one relationship between the system specification and the analysis model components and their interaction. Second, MPA

models themselves are parallel but are evaluated in a sequential manner, due to their sequential implementation as Matlab scripts.

The proposed approach splits the model generation into two phases. First, an analysis metamodel is generated representing the data dependencies of actors in the dataflow process network and the mapping of the application onto an architecture. The metamodel contains only elements that are common to formal performance analysis methods and is thus independent of any specific analysis method. In the second phase, a code generator refines the metamodel using method-specific abstractions and generates the corresponding code.

These two phases are now described in the context of the DOL design flow. After DOL specifications have been translated into the analysis metamodel, code for the Matlab MPA toolbox [Wandeler and Thiele 2006c] is created based on this model. Note that the modular structure of the model generation allows for a simple extension towards a generic generation principle. At the front end, different kinds of system specifications could be translated into the analysis metamodel. At the back end, different code generators could be implemented to target other analysis methods, such as SymTA/S, MAST, or timed automata. The viability of such an approach has been demonstrated in Perathoner et al. [2009], in which several systems have been manually modeled using these analysis methods for comparison purposes. Also note that only the front end needs to be adjusted to extend the framework to a new MPSoC platform (at least as long as this platform is in the modeling scope of the back-end analysis method).

6.1. Metamodel Generation

The first phase of model generation is to transform a DOL specification consisting of an application, architecture, and mapping specification into the analysis metamodel, which is saved as an XML file, as shown in Listing 4. This model contains all the information required for analyzing the performance of a system, afterwards. The proposed metamodel can be viewed as a directed graph $\mathcal{M} = (V, E)$. The set of vertices V is a union $V_P \bigcup V_{\text{AND}}$, where V_P is the set of processing components to model computation and communication, and V_{AND} is the set of abstract AND components [Haid and Thiele 2007; Jersak et al. 2005] for modeling the activation scheme of processing components. (We use AND components for illustration purposes in this article. Abstract OR components could be applied similarly, depending on the semantics of an actor). The set of edges E represents event streams (`connection` elements in XML). All the elements of $\mathcal{M}$ can be annotated with parameters relevant for performance analysis. In particular, the mapping is conveyed by annotating processing components with the necessary binding and scheduling parameters; see lines 9–14 in Listing 4.

Listing 4. Snippet of analysis metamodel XML.

```
 1 <pjd name="P1_trigger" period="200000" jitter="1000000"/>
 2 <sink name="P3_sink"/>
 3 <task name="P1" bcet="27487" wcet="29668"
 4   workload_lower="27487;54984;82481;110033;139359;167842"
 5   workload_upper="29668;58096;86569;115042;143515;171982"/>
 6 ...
 7 <connection from="P1" to="C1"/>
 8 <connection from="C1" to="P2"/>
 9 <resource name="processor_1" bandwidth="1.0000">
10   <fp preemptive="true">
11     <priority value="1"><binding task="P3"/></priority>
12     <priority value="2"><binding task="P1"/></priority>
13   </fp>
14 </resource>
15 ...
```

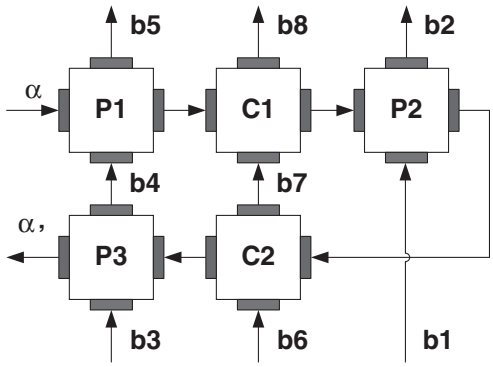

Fig. 8. MPA model with nested cyclic dependencies.

To construct metamodel $\mathcal{M}$, an abstract processing component is instantiated for each actor in the dataflow process network. Processes with multiple inputs are modeled by inserting abstract AND components. The modeling of inter-process communication depends on the binding. If a channel is bound to a shared interconnect, as shown in Figure 7, a corresponding processing component is instantiated. Otherwise, the associated processes are directly connected.

6.2. MPA Code Generation

The second phase of model generation is the creation of a model that is specific to the chosen performance analysis method and is based on the metamodel that has been previously described. More precisely, the metamodel needs to be transformed, such that it can be analyzed by a tool implementing the particular analysis method. In our design flow, the Matlab MPA toolbox [Wandeler and Thiele 2006c] is targeted.

The MPA code generation itself consists of two steps. First, the metamodel is expressed using the MPA specific abstractions. Event streams are replaced by arrival curves; processing components by GPCs; and the mapping information is expressed using service curves, as shown in Figure 3. The result of this first step is a graph whose vertices are GPCs and whose edges are arrival and service curves. In a second step, the sequential Matlab script implementing this graph is generated. Basically, this can be done by performing a topological sort on the graph. This is limited to acyclic directed graphs, but MPA models may contain cycles that either contain event streams (arrival curves) only or both event streams (arrival curves) and resource streams (service curves). In this article, we do not consider the first case that corresponds to feedback loops in the dataflow process network, because their analysis is an active area of research [Thiele and Stoimenov 2009]. An example for the second case is shown in Figure 8, which corresponds to the example shown in Figure 2 with reversed priorities: GPC `C1`, `C2`, and `P2` form an inner cycle because the input resource stream of `C1` depends on `C2`, and the input event stream of `C2` is provided by `C1` via `P2`. All GPCs, together, form the outer cycle, where the input resource stream of `P1` depends on `P3`, and the input event stream of `P3` originates from `P1` and traverses all other GPCs. Note that both priority configurations are reasonable. The configuration shown in Figure 8 ensures that data that is already in the system is handled with higher priority than newly arriving data. In contrast, the configuration in Figure 2(b) ensures that the processing of data starts as soon as it is available. To resolve cyclic dependencies, fixed-point iterations are applied. The correctness of the approach has been formally proven in Jonsson et al. [2008].

Algorithm 1 Greedy algorithm for Matlab code generation.

1: let $V' = V_P$ ▷ V_P is the set of all processing components
2: let $V'' \subset V_P$: $\forall v \in V''$ both event and resource streams are available
3: **procedure** GREEDY_GEN(V', V'')
4: **while** $V' \neq \emptyset$ **do**
5: **while** $V'' \neq \emptyset$ **do**
6: get $g_{i,j} \in V''$; generate Matlab code with ($\alpha_{g_{i,j}}$, $\beta_{g_{i,j}}$)
7: **if** $\exists g_{i+1,j}$ **then** ▷ $g_{i+1,j}$ is the successor of $g_{i,j}$ with respect to β
8: mark $\beta_{g_{i+1,j}}$ available
9: **if** $\alpha_{g_i+1,j}$ available **then** $V'' = V'' \bigcup \{g_{i+1,j}\}$ **end if**
10: **end if**
11: **if** $\exists g_{i,j+1}$ **then** ▷ $g_{i,j+1}$ is the successor of $g_{i,j}$ with respect to α
12: mark $\alpha_{g_{i,j+1}}$ available
13: **if** $\beta_{g_i,j+1}$ available **then** $V'' = V'' \bigcup \{g_{i,j+1}\}$ **end if**
14: **end if**
15: $V'' = V'' \setminus \{g_{i,j}\}$; $V' = V' \setminus \{g_{i,j}\}$
16: **end while**
17: **if** $V' \neq \emptyset$ **then** ▷ cyclic dependency detected
18: get $g_{i,j}$ whose event stream is available
19: find $g_{i,j+x}$ whose resource stream is available
20: $\beta_{g_{i,j}} = \beta_{g_{i,j+x}}$
21: $V'' = V'' \bigcup \{g_{i,j}\}$
22: construct fixed point iteration procedure for ($\alpha_{g_{i,j}}$, $\beta_{g_{i,j+x}}$)
23: call GREEDY_GEN(V', V'') ▷ recursively create code for inner cycles
24: **end if**
25: **end while**
26: **end procedure**

Listing 5. Matlab script for the MPA model in Figure 8 exhibiting cyclic dependencies.

```
1  % initialize all variables
2  b4 = b3;
3  b4_upper = b4(1); b4_lower = b4(2);
4  b4_upper_last = rtcuplus(b4_upper); b4_lower_last = rtcuplus(b4_lower);
5  conv_b4 = false;
6  for it_b4 = (1:MAX_ITERATIONS)
7    [ P1_out b5 P1_delay P1_buf ] = rtcpgc(P1_src_out, b4, P1_demand);
8    b7 = b6;
9    b7_upper = b7(1); b7_lower = b7(2);
10   b7_upper_last = rtcuplus(b7_upper); b7_lower_last = rtcuplus(b7_lower);
11   conv_b7 = false;
12   for it_b7 = (1:MAX_ITERATIONS)
13     [ C1_out b8 C1_delay C1_buf ] = rtcpgc(P1_out, b7, C1_demand);
14     [ P2_out b2 P2_delay P2_buf ] = rtcpgc(C1_out, b1, P2_demand);
15     [ C2_out b7 C2_delay C2_buf ] = rtcpgc(P2_out, b6, C2_demand);
16     b7_upper = b7(1); b7_lower = b7(2);
17     if ((b7_upper == b7_upper_last) && (b7_lower == b7_lower_last))
18       conv_b7 = true; break;
19     end
20     b7_upper_last = rtcuplus(b7_upper);
21     b7_lower_last = rtcuplus(b7_lower);
22   end
23   % if fixed point was not reached, report error and exit
24   [ P3_out b4 P3_delay P3_buf ] = rtcpgc(C2_out, b3, P3_demand);
25   b4_upper = b4(1);  b4_lower = b4(2);
26   if ((b4_upper == b4_upper_last) && (b4_lower == b4_lower_last))
27     conv_b4 = true; break;
28   end
29   b4_upper_last = rtcuplus(b4_upper); b4_lower_last = rtcuplus(b4_lower);
30 end
31 % if fixed point was not reached, report error and exit
```

The pseudocode of the greedy algorithm used for model generation is shown in Algorithm 1. First, the set V'' is computed containing those GPCs whose arrival curves and service curves are both available. Then, iteratively, a random GPC is chosen from V'', and the corresponding Matlab statement is generated (line 6). Afterwards, the output arrival and service curve are marked to be available (lines 8 and 12, respectively), and subsequent GPCs are added to V'' if both their arrival and service curves become available. If a cyclic dependency is detected (line 17), a fixed-point iteration routine is constructed. Within this routine, cycles are resolved by recursively calling

the algorithm (line 23). The algorithm continues until the Matlab statements for all GPCs have been generated.

As an example, the Matlab script generated by applying Algorithm 1 to the MPA model with nested cyclic dependencies in Figure 8 is shown in Listing 5. Lines 12–23 and 6–31 resolve the inner (`C1`, `P2`, and `C2`) and outer (all GPCs involved) cyclic dependencies, respectively. The Matlab statements for the GPCs `P1`, `C1`, `P2`, `C2`, and `P3` are in lines 7, 13–15, and 24, respectively.

7. ANALYSIS MODEL CALIBRATION

In a compositional approach, the goal of model calibration is the determination of bounds on the best-case and worst-case behavior of all system components and the environment. For some parameters, this is a simple task. The timing behavior of the input stream(s), the clock frequencies of the architectural components, and scheduling parameters can be directly taken from the application, architecture, and mapping specification, respectively. These parameters are directly reflected in the input arrival curves and the service curves of an MPA model. The calibration of the GPCs modeling the application is more involved. The workload curves γ, data consumption curves ρ, and data production curves π depend, in general, on the processed data stream, the architecture, and the mapping of the application onto the architecture.

One way to obtain the parameters for formal system-level performance analysis are formal *component-level* analysis methods. Formal methods provide safe bounds on component behavior, such as the worst-case and best-case execution time of an actor [Wilhelm et al. 2008] or the workload of an actor [Maxiaguine et al. 2004]. Unfortunately, this class of methods suffers from a restricted scope and often cannot be completely automated. Manually providing the required information increases the overall modeling effort and decreases the level of automation. The main advantage of formal methods is their safeness, that is, worst-case or best-case bounds can be guaranteed, and, therefore, the performance analysis results can be used for the validation of hard real-time systems.

An alternative approach is simulation. Calibration by simulation cannot guarantee safe bounds unless exhaustive test patterns are used. Because the complexity of the application and architecture often prohibits exhaustive simulation of the entire system, a compositional approach, as advocated in this article, needs to be applied. This means that the individual (simple) components are exhaustively simulated in isolation, and the (complex) global interferences and interactions are analyzed formally by means of a modular method, such as MPA.

In addition, calibration by simulation can be used to estimate bounds for a restricted set of behaviors, as characterized by the variability curves (γ, ρ, π). This is facilitated by two properties of the variability curves used in MPA. First, the curves obtained by a single simulation run do not only bound the specific behavior observed in that run but cover an entire set of behaviors. Second, results of multiple simulation runs can be easily combined by taking the minimum and maximum of the curves obtained in different runs. Compared to formal methods, simulation usually provides tighter (but possibly unsafe) bounds and can be performed completely automated. Due to these reasons, we use simulation for calibration in the experimental section of this article. Following, some of the practical details of this approach are discussed.

7.1. Calibration Using Functional and Timed Simulation

The quantities that need to be obtained by simulation are a) workload curves of computation and communication tasks γ, b) data production curves π, and c) data consumption curves ρ. To create these curves, the accumulated workload and the accumulated number of read-and-write accesses to each channel are logged during simulation for

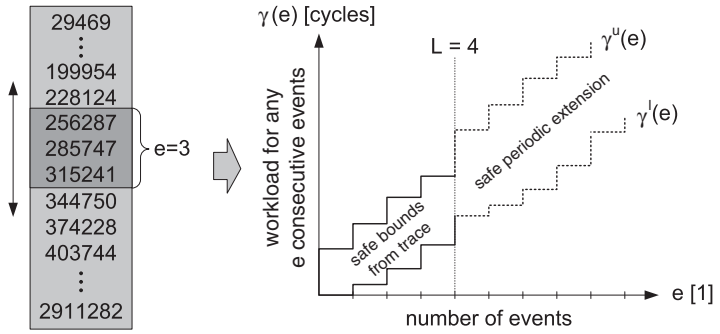

Fig. 9. Construction of workload curve from trace of accumulated workloads.

each individual actor. After simulation, a trace t of length N is available for each of these quantities. We assume that the traces contain safe upper and lower bounds on the workload, data production, and data consumption for up to L consecutive actor activations.

Based on this assumption, safe upper and lower bounds of the workload for up to L consecutive events can be obtained by sliding windows of size 1 to L across the trace of accumulated workloads, such that

$$\gamma^u(e) = \max_{0 \le n < N-e} \{t[n+e] - t[n]\} \qquad 0 \le e \le L, \tag{14}$$

$$\gamma^l(e) = \min_{0 \le n < N-e} \{t[n+e] - t[n]\} \qquad 0 \le e \le L. \tag{15}$$

These variability curves can be safely extended to intervals greater than L by assuming that all best-case and worst-case quantities for intervals up to L activations in any trace have been encountered. Every interval larger than L can now be partitioned into $\lfloor e/L \rfloor$ intervals of length L and the remaining length $e \,\%\, L$. As a result, safe bounds for intervals greater than L can be derived by periodically extending the obtained curve segments, as shown in Figure 9. Note that even though the bounds are periodically extended, this does not imply that a stream bounded by these curves needs to be periodic.

$$\gamma(e) = \lfloor e/L \rfloor \cdot \gamma(L) + \gamma(e \,\%\, L) \qquad e > L. \tag{16}$$

Similarly, data production curves π and data consumption curves ρ can be constructed.

Due to the determinism of dataflow process networks, traces of timing-independent parameters can be obtained from functional simulation. In particular, this applies to the workload curves of communication tasks, data production curves, and data consumption curves. The workload curves for actors are obtained from timed simulation. In the following experiments, estimations for the workload curves of actors, as well as context-switch times, are determined based on traces obtained by timed simulation on MPARM. As mentioned already, safe bounds could be obtained by leveraging formal component-level analysis methods.

8. EXPERIMENTS

In this section, we use a prototype implementation of the DOL design flow to analyze three different applications: a producer-consumer (P-C) example, and two streaming applications, namely a Motion-JPEG (MJPEG) decoder and a wave field synthesis (WFS) application. All these applications run on top of the MPARM [Benini et al. 2005] cycle-accurate simulator.

Table I. Java Code Size of Different Parts of the DOL Design Flow

Part of design flow	Lines of code
DOL representation of system specifications	6200
functional simulation generator	4100
MPARM code generator	2100
analysis meta-model generator	700
log-file analysis of functional and timed simulation	1200
MPA Matlab script generator	4300

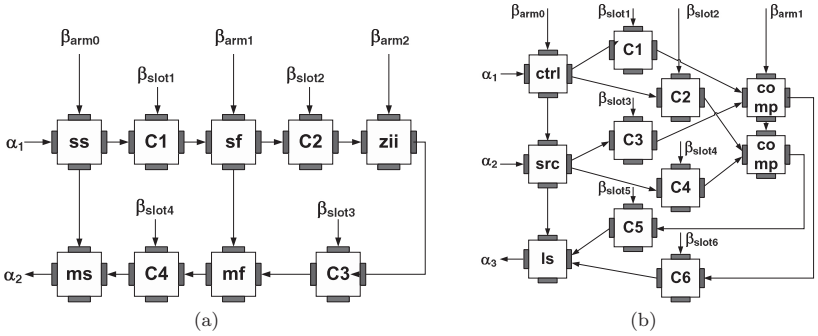

Fig. 10. MPA models of case study applications. (a) 5-stage MJPEG application mapped onto a 3-tile MPARM platform. (b) WFS application mapped onto a 2-tile MPARM platform.

The DOL and the analysis model generation have been implemented in Java. The packages are available for download at http://www.tik.ee.ethz.ch/∼shapes. To give an indication about the size of the prototype implementation, Table I shows the code size of different parts of the implementation.

8.1. Case Study Applications

Producer-Consumer. The producer-consumer example is depicted in Figure 2(a) and its corresponding MPA model in Figure 2(b). The producer `P1` generates a stream of floating point numbers, which pass through `P2` and are consumed by `P3`.

MJPEG Decoder. The MJPEG decoder decompresses a movie stream by applying JPEG decompression to each individual frame. Because of the inherent parallelism in the JPEG algorithm, the decoder executes in a pipelined fashion with five stages, each implemented by an actor, as shown in Figure 10(a). The first and last stages are the splitting of a stream into frames (ss) and the merging of frames back to a stream (ms), respectively. The variable length decoding and splitting of frames into macroblocks forms the second stage (sf). The zigzag scan, inverse quantization, and the inverse discrete cosine transform form the third stage (zii). Combining macroblocks back to frames forms the fourth stage (mf). The application is mapped onto a 3-tile MPARM system, resulting in the MPA model depicted in Figure 10(a). The used bitstream consists of 31 frames encoded in the QVGA (320×240) YUV 444 format. In the used implementation, control parameters, Huffman and quantization tables, as well as image blocks, are transmitted over the same channels, resulting in variable consumption and production rates for actors.

Wave Field Synthesis. By using an array of loudspeakers and audio beamforming techniques, WFS allows for reproducing an acoustic sound field whose perceived origin is not restricted to the position of physical loudspeaker boxes. The MPA model of a WFS application that renders a sound source using 16 loudspeakers and that is mapped onto a 2-tile MPARM system is shown in Figure 10(b). The *source* actor reads a (mono) audio signal and the *control* actor reads the beamforming coefficients from a digital

Table II. Duration of Analysis Model Generation and Calibration

Step		Duration		
		P–C	MJPEG	WFS
model calibration (one-time effort)	functional simulation generation	22 s	42 s	35 s
	functional simulation	0.2 s	3.6 s	2.4 s
	synthesis (generation of binary)	2 s	4 s	3 s
	simulation on MPARM	23 s	13550 s	740 s
	log-file analysis and back-annotation	1 s	12 s	3 s
model generation		1 s	1 s	1 s
performance analysis based on generated model		0.2 s	2.5 s	1.4 s

Note: Measured on a 1.86 GHz Intel Pentium Mobile machine with 1 GB of RAM.

interface. The signal processing takes place in the *compute* actors, whereby each actor synthesizes the signals for eight channels. The computed signals are communicated to the *loudspeaker* actor, which drives the D/A converters. The processing is done in tokens of 32 samples, each represented as a single-precision 32-bit floating point number. The sampling rate is 48 kHz.

8.2. Results

The experiments are used to evaluate the proposed approach with respect to the time needed to obtain results (efficiency) and the quality of the obtained bounds.

Efficiency. Table II lists the durations of the single steps to generate, calibrate, and evaluate the MPA models of the producer-consumer, MJPEG, and WFS application.

The table shows that calibration is always the step that takes by far the most time, whereas model generation and performance analysis take a matter of seconds. Based on Table II, one can draw the following conclusions.

First, evaluating a system's performance using cycle-accurate simulation is by several orders of magnitude slower than formal performance analysis. This justifies the proposed approach of integrating a formal performance analysis in the design cycle, use formal performance analysis during design space exploration, and restrict the usage of cycle-accurate simulation to the calibration of the formal model.

Second, depending on the complexity of the application, the elapsed simulation time varies in a range from seconds to hours. Contrarily, the time used for generating the analysis model and for performing the analysis are almost identical for all the applications. This again shows the importance of integrating formal performance analysis into the MPSoC design cycle.

Third, even though calibration is completely automated, it takes a considerable time. Thus, manual calibration at the same level of detail would be a very difficult and a time-consuming endeavor. Similar observations can be made with respect to model generation. This supports our claim that automated model generation and calibration are essential when compositional performance analysis is applied for analyzing real systems.

Fourth, calibrating the system model for each new binding and scheduling alternative during design space exploration would be prohibitively slow, mainly due to the long runtime of the timed simulation. Due to the compositional approach, however, one can collect all the parameters prior to design space exploration.

Table III shows the size of the generated Matlab scripts for all three considered applications. One can draw the following conclusions. As the complexity of the system grows, the analysis model is more involved, and the size of the model is accordingly larger. In addition, when fixed-point iteration needs to be applied because of cyclic resource dependencies, the size of the generated Matlab script increases considerably.

Table III. Code Size of Matlab Scripts Generated for the Case Study Applications

Application		Without cyclic dep.	With cyclic dep.
P–C	lines of code	77	99
	code size (bytes)	3204	3780
MJPEG	lines of code	117	161
	code size (bytes)	15689	16905
WFS	lines of code	116	138
	code size (bytes)	7725	8301

Note: The column labeled "without cyclic dep." refers to the MPA models depicted in Figure 2(b) and Figure 10. The column labeled "with cyclic dep." refers to the same MPA models but with reversed priorities. Refer to Table IV for details about the two different configurations.

Table IV. Comparison of Worst-Case Values Observed During Simulation and Bounds Computed Using MPA

			Simulation		MPA		MPA†			Simulation		MPA	
Actor	proc.	pr.	del.	buf.	del.	buf.	del.	buf.	pr.	del.	buf.	del.	buf.
p–c.p1	1	1	143	4	208	5	280	5	2	313	4	534	6
p–c.p3	1	2	276	5	363	5	788	9	1	36	1	48	1
p–c.p2	2	1	102	2	143	3	285	4	1	68	2	150	3
mjpeg.ss	1	1	16	2	52	6	148	6	2	23	2	111	6
mjpeg.ms	1	2	80	1	117	4	338	8	1	23	1	38	1
mjpeg.sf	2	1	266	5	287	6	388	6	2	344	5	465	6
mjpeg.mf	2	2	298	3	488	4	734	7	1	36	1	51	1
mjpeg.zii	3	1	695	6	746	6	908	7	1	600	5	812	7
wfs.ctrl	1	1	78	2	109	2	152	2	3	294	2	590	3
wfs.src	1	2	199	3	219	3	280	3	2	193	2	279	2
wfs.ls	1	3	2465	6	5357	10	5697	11	1	2049	4	5861	8
wfs.comp1	2	1	1999	7	3389	7	3472	7	2	3123	10	6148	15
wfs.comp2	2	2	3619	13	4895	13	5049	13	1	1872	6	4122	9

Note: The column labeled MPA† shows the bounds when using best-/worst-case execution times for actor characterization, whereas the other (tighter) bounds are computed using workload curves. Delays (del.) are given in 10^3 processor cycles and backlogs (buf.) in tokens. The table shows results for two mappings of each application whereby the MPA model of the right one exhibits cyclic dependencies. The columns labeled "proc." and "pr." indicate the processor to which an actor is bound and its priority.

Clearly, manually constructing analysis models for such systems becomes error-prone, and, therefore, automated generation is desirable.

Quality. Table IV compares several bounds derived by MPA to the actual (average-case behavior) quantities observed during simulation of the considered systems. Note that an accurate, quantitative evaluation of the quality (tightness) of the bounds would require exhaustive simulation of these systems covering all corner cases. Unfortunately, this is not feasible, because finding and simulating exhaustive test patterns that cover all possible corner cases and interference through joint resources takes too much time. Still, it can be argued that the bounds are in a reasonable range typical for compositional performance analysis. Differences in the same range have been observed for several systems, for instance, in Perathoner et al. [2009]. There are two main reasons for these differences. The first is that several operators in the formal performance analysis do not yield tight bounds. The second is that the timed simulations only exhibit the worst-case and best-case behavior at component-level but not at system level. Finally, note that for the metrics in Table IV, MPA can only be used to provide upper bounds. If other metrics such as resource utilization or throughput were considered, lower bounds could also be computed using MPA.

9. CONCLUSIONS

The distributed operation layer (DOL) MPSoC software design flow targeted at the development of real-time streaming applications has been presented in this article. For

applications expressed as dataflow process networks, this design flow allows for the automatic creation of implementation, as well as a formal performance analysis model for system validation. Because the same system specification is used as the basis for software synthesis and for generating the analysis model, the gap between high-level analysis model and system implementation is kept small. By calibrating the generated model using simulation, an analysis model can be generated that faithfully models the real system, such that accurate best-case/worst-case bounds for system properties can be obtained. The effectiveness of the proposed design flow has been illustrated for a video (MJPEG) and an audio application (wave field synthesis) executing on a multiprocessor ARM architecture.

REFERENCES

Balsamo, S., Marco, A. D., Inverardi, P., and Simeoni, M. 2004. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng. 30,* 5, 295–310.

Benini, L., Bertozzi, D., Alessandro, B., Menichelli, F., and Olivieri, M. 2005. MPARM: Exploring the multi-processor soc design space with SystemC. *J. VLSI Signal Process. 41*, 169–182.

Black, B. and Shen, J. P. 1998. Calibration of microprocessor performance models. *Comput. 31,* 5, 59–65.

Chakraborty, S., Künzli, S., and Thiele, L. 2003. A general framework for analyzing system properties in platform-based embedded system design. In *Proceedings of the Design, Automation and Test in Europe (DATE)*. 190–195.

Densmore, D., Sangiovanni-Vincentelli, A., and Passerone, R. 2006. A platform-based taxonomy for ESL design. *IEEE Design Test Comput. 23,* 5, 359–374.

Gerstlauer, A., Haubelt, C., Pimentel, A. D., Stefanov, T., Gajski, D. D., and Teich, J. 2009. Electronic system-level synthesis methodologies. *IEEE Trans. Comput.-Aid. Design Integr. Circuits Syst. 28,* 10, 1517–1530.

González Harbour, M., Gutiérrez García, J. J., Palencia Gutiérrez, J. C., and Drake Moyano, J. M. 2001. MAST: Modeling and analysis suite for real time applications. In *Proceedings of the Euromicro Conference on Real-Time Systems*. 125–134.

Haid, W., Keller, M., Huang, K., Bacivarov, I., and Thiele, L. 2009. Generation and calibration of compositional performance analysis models for multiprocessor systems. In *Proceedings of the International Conference on Systems, Architectures, Modeling and Simulation (IC-SAMOS)*. 92–99.

Haid, W. and Thiele, L. 2007. Complex task activation schemes in system level performance analysis. In *Proceedings of the International Conference on HW/SW Codesign and System Synthesis (CODES/ISSS)*. 173–178.

Henia, R., Hamann, A., Jersak, M., Racu, R., Richter, K., and Ernst, R. 2005. System level performance analysis — The SymTA/S approach. *IEE Proc.: Comput. Digital Tech. 152,* 2, 148–166.

Jersak, M., Richter, K., and Ernst, R. 2005. Performance analysis for complex embedded systems. *Int. J. Embedd. Syst. 1,* 1–2, 33–49.

Jonsson, B., Perathoner, S., Thiele, L., and Yi, W. 2008. Cyclic dependencies in modular performance analysis. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*. 179–188.

Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., and Shippy, D. 2005. Introduction to the cell multi-processor. *IBM J. Res. Develop. 49,* 4/5, 589–604.

Kahn, G. 1974. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*. 471–475.

Kangas, T., Kukkala, P., Orsila, H., Salminen, E., Hännikäinen, M., and Hämäläinen, T. D. 2006. UML-based multiprocessor soc design framework. *ACM Trans. Embedd. Comput. Syst. 5,* 2, 281–320.

Künzli, S., Poletti, F., Benini, L., and Thiele, L. 2006. Combining simulation and formal methods for system-level performance analysis. In *Proceedings of the Design, Automation and Test in Europe (DATE)*. 236–241.

Lahiri, K., Raghunathan, A., and Dey, S. 2001. System-level performance analysis for designing on-chip communication architectures. *IEEE Trans. Comput.-Aid. Design Integr. Circuits Syst. 20,* 6, 768–783.

Lampka, K., Perathoner, S., and Thiele, L. 2010. Analytic real-time analysis and timed automata: A hybrid methodology for the performance analysis of embedded real-time systems. *Des. Autom. Embedd. Syst. 14,* 3, 193–227.

Le Boudec, J.-Y. and Thiran, P. 2001. *Network Calculus — A Theory of Deterministic Queuing Systems for the Internet*. Lecture Notes in Computer Science, vol. 2050. Springer-Verlag, Berlin, Germany.

LEE, E. A. AND MESSERSCHMITT, D. G. 1987. Synchronous data flow. *Proc. IEEE 75,* 9, 1235–1245.

LEE, E. A. AND PARKS, T. M. 1995. Dataflow Process Networks. *Proc. IEEE 83,* 5, 773–799.

MAXIAGUINE, A., KÜNZLI, S., AND THIELE, L. 2004. Workload characterization model for tasks with variable execution demand. In *Proceedings of the Design, Automation and Test in Europe (DATE)*. 1040–1045.

PAOLUCCI, P., JERRAYA, A., LEUPERS, R., THIELE, L., AND VICINI, P. 2006. SHAPES: A tiled scalable software hardware architecture platform for embedded systems. In *Proceedings of the International Conference HW/SW Codesign and System Synthesis (CODES/ISSS)*. 167–172.

PELLIZZONI, R. AND CACCAMO, M. 2007. Toward the predictable integration of real-time COTS-based systems. In *Proceedings of the Real-Time Systems Symposium (RTSS)*. 73–82.

PERATHONER, S., REIN, T., THIELE, L., LAMPKA, K., AND ROX, J. 2010. Modeling structured event streams in system level performance analysis. In *Proceedings of the ACM Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*.

PERATHONER, S., WANDELER, E., THIELE, L., HAMANN, A., SCHLIECKER, S., HENIA, R., RACU, R., ERNST, R., AND GONZÁLEZ HARBOUR, M. 2009. Influence of different abstractions on the performance analysis of distributed hard real-time systems. *Des. Autom. Embedd. Syst. 13,* 1, 27–49.

PETRIU, D., SHOUSHA, C., AND JALNAPURKAR, A. 2000. Architecture-based performance analysis applied to a telecommunication system. *IEEE Trans. Softw. Eng. 26,* 11, 1049–1065.

PIMENTEL, A., ERBAS, C., AND POLSTRA, S. 2006. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput. 55,* 2, 99–112.

PIMENTEL, A. D., THOMPSON, M., POLSTRA, S., AND ERBAS, C. 2008. Calibration of abstract performance models for system system-level design space exploration. *J. Signal Process. Syst. 50,* 2, 99–114.

RTEMS STEERING COMMITTEE. 2010. RTEMS. http://www.rtems.com.

THIELE, L., BACIVAROV, I., HAID, W., AND HUANG, K. 2007. Mapping-applications to tiled multiprocessor embedded systems. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD)*. 29–40.

THIELE, L. AND STOIMENOV, N. 2009. Modular performance analysis of cyclic dataflow graphs. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*. 127–136.

THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. 2002. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*. 179–196.

VIEHL, A., SCHÖNWALD, T., BRINGMANN, O., AND ROSENSTIEHL, W. 2006. Formal performance analysis and simulation of UML/SysML models for ESL design. In *Proceedings of the Design, Automation and Test in Europe (DATE)*. 242–247.

WANDELER, E. AND THIELE, L. 2006a. Interface-based design of real-time systems with hierarchical scheduling. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 243–252.

WANDELER, E. AND THIELE, L. 2006b. Optimal TDMA time slot and cycle length allocation for hard real-time systems. In *Proceedings of the Asia and South Pacific Conference on Design Automation (ASP-DAC)*. 479–484.

WANDELER, E. AND THIELE, L. 2006c. Real-Time Calculus (RTC) toolbox. http://www.mpa.ethz.ch/Rtctoolbox.

WANDELER, E., THIELE, L., VERHOEF, M., AND LIEVERSE, P. 2006. System architecture evaluation using modular performance analysis: A case study. *Int. J. Softw. Tools Technol. Transfer 8,* 6, 649–667.

WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. 2008. The worst-case execution time problem — Overview of methods and survey of tools. *ACM Trans. Embedd. Comput. Syst. 7,* 3, 36:1–36:53.

WOODSIDE, M. 2007. *From annotated software designs (UML SPT/MARTE) to model formalisms*. Lecture Notes in Computer Science, vol. 4486. Springer-Verlag, Berlin, Germany, 429–467.

Received February 2009; revised January, October 2010; accepted October 2010