Abstracting Functionality for Modular Performance Analysis of Hard Real-Time Systems

Ernesto Wandeler Lothar Thiele
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zurich, Switzerland
E-mail: {wandeler,thiele}@tik.ee.ethz.ch

Abstract— System level performance analysis techniques play an important role in the design process of complex embedded systems. They allow to analyze essential characteristics of a system design in an early design stage and support therewith the choice of important design decisions. While analytical methods for system level performance analysis lead to hard bounded analysis results, the obtained results are often overly pessimistic due to a lack of details such analytical methods can incorporate in their system analysis. To overcome this problem, we present new abstract models for event streams and system components of embedded systems, and show how these models can be combined to modules for modular performance analysis. With the presented models, we can capture complex functional properties of systems, as for example caches, variable resource demand of events in an event stream, or arbitrary up- and down-sampling of event streams in a system component. The applicability of our models and their advantages over traditional models for performance analysis are shown in a case study of a system component with LRU (Least Recently Used) cache.

I. Introduction

Complex real-time embedded systems are often comprised of a combination of different hardware and software components that are triggered by incoming event streams and that communicate via some communication network. Such systems are often integrated as a System on Chip (SoC) and many alternatives for partitioning, allocation and binding lead to a large design space.

During the system level design process of such a complex system, a designer is typically faced with the questions whether the system-level timing properties of a certain implementation will meet the design requirements, what the different bus utilizations will be, which bus or processor acts as a bottleneck or what the memory requirements will be. One of the major challenges in the design process is therefore to analyze these essential characteristics of system in an early design stage, to support the choice of important design decisions before much time must be invested in detailed implementations.

Obtaining tight results for the above mentioned characteristics of a system design is not only difficult because of the heterogeneity and complexity of the complete system, but also because of the possibly complex behaviors of single components in the system. Different events may arrive on incoming event streams of a component and they may create different resource demands depending on their event type, see e.g. [6]. And moreover, the resource demand an event creates in a component may not only depend on its type, but also on the internal state of the component, for example due to caching effects.

Currently, the analysis of such complex and heterogeneous systems is mainly based on simulation, using for example SystemC, or trace based simulation as in [10]. These techniques allow modeling of complex systems and complex component behaviors in any level of detail. However, in terms of completeness, simulation based approaches do not allow to get worst-case results, because any concrete simulation-run can in general not guarantee to cover all corner cases. But most of all, simulation often suffers from long run-times and from a high set-up effort for each new architecture and mapping to be analyzed and is therefore not well suited for performance analysis in an early design stage.

In contrast to simulation, formal analytical methods allow to obtain the hard bounded results that are needed for the analysis of hard real-time systems. A well-known technique to model real-time components are Timed Automata [1]. In [5] it was shown that timed automata can be used as task models for event-driven systems and that the schedulability problem in such a model can be transformed to a reachability problem for timed automata and is thus decidable. While we could model complex component behaviors in any detail using this technique, the limitation to synchronous communication in timed automata would require to explicitly model buffers in a stream based application with asynchronous communication. This however would usually turn the analysis effort for a complete system to be prohibitive.

On the other hand, powerful abstractions have been developed in the domain of communication networks, to model flow of data through a network. In particular the theoretical framework called Network Calculus [9] provides a means to deterministically reason about timing properties of data flows in queueing networks. Real-Time Calculus [14] extends the basic concepts of Network Calculus to the domain of real-time embedded systems and in [2] a unifying approach to system level performance analysis with Real-Time Calculus has been proposed. It is based on a general event model, allows for hierarchical scheduling and arbitration, and can take computation and communication resources into account. But the models used in this framework cannot take into account the complex behaviors of single system components. Instead, a component is characterized only by the largest possible worst-case execution time (WCET) and the smallest possible best-case execution time (BCET) any event could have. While this framework can be used for hard real-time analysis of a system, it will in often lead to overly pessimistic results and therefore to unnecessary expensive system designs.

Another analytical method for system level performance analysis was proposed in [13] and is based on classical scheduling results from hard real-time system design. This method uses a number of well known abstractions of the timing behavior of event streams and provides additional interfaces between them. But this method also suffers from overly pessimistic results as complex system behaviors are not taken into account.

Recently, some methods were presented, which try to address this problem at least partially. In [7], a performance analysis technique was proposed that uses *system contexts* to improve the analysis results. The system contexts represent various kinds of correlations in task execution sequences within a system component and it was shown that exploiting knowledge about the structure of correlated event streams may result in improved analysis results. However, the methods and models described in [7] are confined to particular problem instances and cannot be applied in a general context.

In [15], *type rate curves* were proposed as an abstract model for event streams. Type rate curves can capture information of possible correlations and dependencies between the different event types on an event stream by bounding the number of occurrences of certain event types in an event stream sequence of given length. This model was used for system level performance analysis and it was shown that it may result in improved analysis results. However, this model cannot capture information about explicit event sequences or system components. For example, it is not possible to capture behaviors of system components that depend on the internal state of a component, like caching. Therefore, the analysis results are often still overly pessimistic.

Contributions

- We present new models for event streams and functional system components of embedded real-time systems, that allow to capture abstract properties that enable improved performance analysis.
- We show methods how these models can be combined to modules and how the captured information can therewith be incorporated into modular system level performance analysis. With the presented models and methods, we can capture for example caches, complex resource demand dependencies of different event types in an event stream, or arbitrary up- and downsampling of event streams in a system component.
- We show the applicability of the presented models and methods in a case study with a system component with an internal LRU cache. The analysis results for typical design problems obtained using the presented models are considerably improved compared to the results obtained using existing methods. The improvements thereby stem from the ability of the presented models and methods to capture the internal LRU cache and to incorporate it into the performance analysis.

II. Modular Performance Analysis

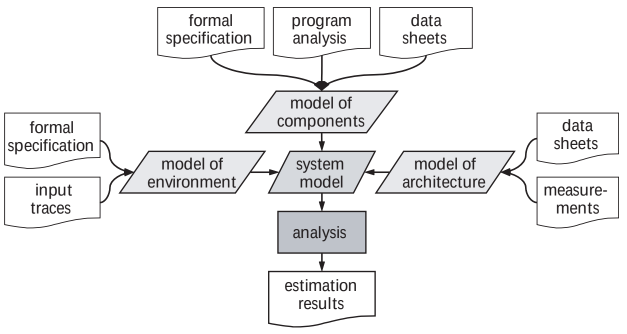

Fig. 1. Elements of modular performance analysis and methods to obtain them.

Figure 1 shows the overview of an approach to modular system level performance analysis of embedded systems. In a first step, appropriate abstract models for the environment, the architecture as well as the functional components must be defined. These models must be able to capture the properties of the different system elements such that they can be used for analytic performance analysis. Further, we need methods to extract the properties of the different system elements into these models. We may get the needed information for example from formal specifications, data sheets, or simulation and traces. Note, in case we extract the models from simulation and traces, the performance analysis does not guarantee hard-real time results anymore, but the obtained results may still be of interest for the analysis of soft real-time systems.

After obtaining abstract models for all system elements, we need to build modules that define the interaction of the different element models and that can be composed to build an abstract model of the complete system. And finally, we need analysis methods that can be used to analyze the abstract system model and that deliver as tight results as possible for the essential characteristics of the concrete system.

The following sections define all elements and methods in Fig. 1 as needed for our proposed performance analysis. In Sect. III, we define abstract models for event streams which describe the system environment, for resource units which describe architectural elements as well as for functional units which describe the functional components of a system. In Sect. IV, we then show how to build modules using these models. And finally, in Sect. V we show how to compose the obtained modules to an abstract model of a complete system and how to analyze this system model.

III. Abstract Element Models

A. Event Stream Model

In an *event stream*, every event has a time when it occurs. Moreover, we define a number of different event types for all events in an event stream. We introduce this distinction into different event types because in our system model the activities and resource demand created by an event will depend on its event type.

We capture the abstract timing information of an event stream using the concept of upper and lower *arrival curves* [4].

Definition 1 (Arrival Curves). *Let $R[s,t)$ denote the number of events that arrive on an event stream in the time interval $[s,t)$. Then, R, $\bar{\alpha}^u$ and $\bar{\alpha}^l$ are related to each other by the following inequality*

$$\bar{\alpha}^l(t-s) \leq R[s,t) \leq \bar{\alpha}^u(t-s), \forall s < t \tag{1}$$

where $\bar{\alpha}^l(0) = \bar{\alpha}^u(0) = 0$.

The timing information of standard event models like *periodic*, *periodic with jitter*, *periodic with bursts*, *sporadic* or of any event stream with a known timing behavior can be represented by an appropriate choice of $\bar{\alpha}^u$ and $\bar{\alpha}^l$ [2]. Moreover it is also possible to determine the values of $\bar{\alpha}^u$ and $\bar{\alpha}^l$ corresponding to any given finite length arbitrary event trace, obtained for example from observation or simulation.

Additionally, we model the functional information, i.e. the information of possible event type sequences, of an event stream by a state-transition graph whose transitions are labeled with symbols corresponding to the different event types occurring on the event stream.

Definition 2 (Event Automaton). *An event automaton F_σ is a tuple (S, S^0, Σ, T), where S is a set of states, $S^0 \subseteq S$ is a set of initial states, Σ is a set of event types, and $T \subseteq S \times \Sigma \times S$ is a set of transitions. The system starts in an initial state, and if $s \xrightarrow{\sigma} s'$ then an event σ may occur on the event stream, leading to a state change from s to s'.*

The functional information of an event stream may be obtained from a formal specification of the event stream, or sometimes it may also be possible to discover sequence patterns in a finite length event trace, which can then be extracted into an event automaton.

When we are given the above presented model of an event stream, described by $\bar{\alpha}^u$, $\bar{\alpha}^l$ and F_σ, we know that in *any* time interval of length Δ, at least $\bar{\alpha}^u(\Delta)$ and at most $\bar{\alpha}^l(\Delta)$ number of events arrive on the stream and that the possible sequences of arriving events is limited to valid runs of given length in the event automaton F_σ.

Example 1. *Figure 2 shows the abstract model of an event stream. From the arrival curves (left), we see that the stream can be modeled as having a period $p = 1ms$ and a jitter $j = 400\mu s$, while from the event automaton (right) we know that there may arrive three different types of events,* a, b *and* c, *in an order restricted by the automaton.*

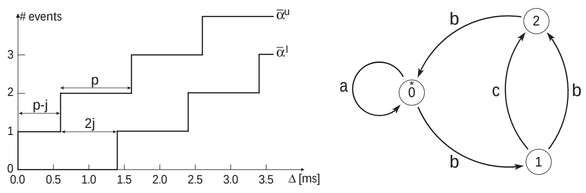

Fig. 2. An abstract event stream. The initial state of F_σ is marked with an asterisk.

B. Resource Model

We characterize the capabilities of a *resource* using lower and upper *service curves* β^l and β^u [9].

DEFINITION 3 (SERVICE CURVES). *Let $C[s,t)$ denote the number of processing or communication units available from a resource over the time interval $[s,t)$, then the inequality*

$$\beta^l(t-s) \leq C[s,t) \leq \beta^u(t-s), \forall s < t \tag{2}$$

holds, where again $\beta^l(0) = \beta^u(0) = 0$..

The service curves of a resource can be determined using data sheets, using analytically derived properties or by using measurements. For example, in the simplest case of an unloaded processor, both the upper and the lower resource curves are equal and are represented by straight lines $\beta^u(\Delta) = \beta^l(\Delta) = f \cdot \Delta$, where f equals the processor speed, i.e. the number of available processing cycles per time unit. We may also model communication resources, where the service curves bound the amount of transmittable bits in a given time interval.

C. Functional Unit Model

In our abstraction, an incoming event stream flows into a FIFO buffer in front of a *functional unit* of an embedded system. The functional unit is then triggered by the different events of this buffered incoming event stream, generates different events on an outgoing event stream and is restricted by the availability of resources. The type of events generated on the outgoing event stream thereby depend on the internal state of the functional unit and on the event types arriving on the incoming event stream. We model a functional unit by a state-transition graph.

DEFINITION 4 (FUNCTIONAL UNIT AUTOMATON). *A functional unit automaton F_U is a tuple $(S, S^0, \Sigma_I, \Sigma_O, D, T)$, where S is a set of states, $S^0 \subseteq S$ is a set of initial states, Σ_I is the non-empty set of accepted incoming event types and Σ_O is the non-empty set of generated outgoing event types. We then define two sets $\Sigma_I^* = \Sigma_I \cup \varepsilon$ and $\Sigma_O^* = \Sigma_O \cup \varepsilon$, that equal the sets of incoming and outgoing event types, both complemented with the empty event ε. Further, D is a demand function $D : S \times \Sigma_I^* \times \Sigma_O^* \times S \rightarrow [\mathbb{Z}^+, \mathbb{Z}^+]$. This function determines the upper and lower bound of the resources needed by a functional unit in order to process an incoming event, generate an outgoing event and change its internal state. Finally, $T \subseteq S \times \Sigma_I^* \times \mathbb{Z}^+ \times \mathbb{Z}^+ \times \Sigma_O^* \times S$ is a set of transitions.*

The system starts in an initial state, and if $s \xrightarrow{\sigma_I/[d_l,d_u]/\sigma_O} s'$ and if the system is triggered by the incoming event σ_I, the functional unit has a resource demand of at least d_l and at most d_u resource units to emit an event σ_O and change its state from s to s'.

We can get the needed information to model a functional unit from a formal specification of the unit or from data sheets. Moreover, it may also be possible to obtain the information from program analysis of the functional unit.

Using the automaton F_U allows us to capture the behavior of a functional unit, and due to the empty event ε, we can also model any kind of resampling and clustering (up-/down-sampling) in the functional unit. We may also model communication units such as buses, where the resource demands express the communication size of different event types. In addition, the model allows to flexibly choose the level of detail for every system component, such that single system components that seem to be critical for the performance of the complete system may be modeled very detailed, while others may be modeled with less details. The simplest automaton F_U would thereby only consist of a single state with a self-loop for every accepted event type.

EXAMPLE 2. *Figure 3 shows the abstract model of a simple functional unit with a LRU cache, which could for example process the event stream in Fig. 2.*

The LRU cache of the modeled functional unit has one cache block that can hold the program code to process either event a *or event* b. *Initially the cache is empty. Whenever an event* a *or* b *arrives and the correct program code is not available in the cache, the code is loaded into the cache and the event is then processed. This generates a resource demand of 10'000 and 15'000 cycles for an event* a *and* b *respectively. If the program code is already in the cache, both events* a *and* b *generate a resource demand of 5'000 cycles.*

The program code to process event c *cannot be loaded into the cache. An arriving event* c *always generates a resource demand of at least 3'000 and at most 20'000 cycles. Additionally, the output generated during processing of an event* c *depends on the internal state of the functional unit; the functional unit generates two events* f *if it is in its initial state (up-sampling) and only one event* g *otherwise.*

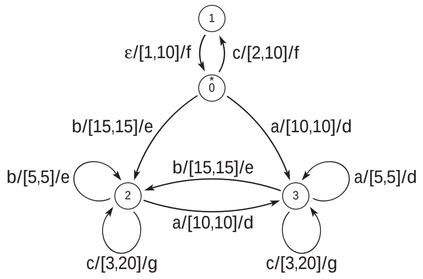

Fig. 3. An abstract functional unit. The numbers of all resource demands are given in 10^3 cycles.

IV. ABSTRACT SYSTEM MODULES

After defining abstract models for all system elements, we need to build abstract system modules using these models. To allow modular performance analysis, these modules must be composable.

Such an abstract module shall take an event stream model, a resource model and a functional unit model as input. To be composable, the module shall then again have an event stream model and a resource model as output. These output models describe the outgoing event stream and the remaining resources after processing the incoming event stream on the functional unit with the available resources. They can then be used as input to other modules, which enables a modular system composition. We build such an abstract system module as depicted in Fig. 4 and we use Real-Time Calculus [14] for performance analysis. Real-Time Calculus also builds the central part of our abstract system module and relates the incoming arrival and service curves to the outgoing arrival and service curves, depicted by the block labeled RTC.

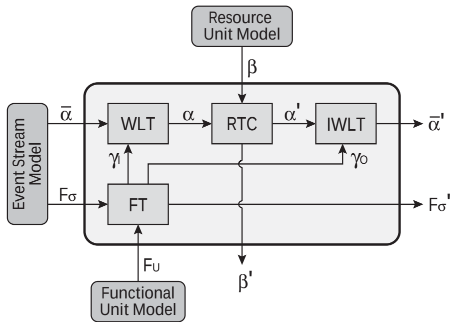

Fig. 4. An abstract system module with abstract models as input and output and internal transformations.

In our models, the service curves of the resource model give an upper and lower bound on the resource capability of a resource, while the arrival curves of an event stream model give an upper and lower bound on the number of arriving events over a given time interval. For performance analysis, we are however not interested in the number of arriving events, but rather in the maximum and minimum resource demand they create over a given time interval on a resource when processed by a functional unit. This information is contained in the functional parts of an event stream model and a functional unit model. We extract this information into so-called input and output workload curves γ_I and γ_O [12], and we then use these workload curves for the workload transformation WLT and the inverse workload transformation IWLT.

All transformations and relations in our abstract system module are described in the following sections.

A. Functional Transformations (FT)

All functional transformations need the product automaton F_{Prod} of the incoming event stream automaton F_σ and the functional unit automaton F_U as a basis. The states of the product automaton are $S_{Prod} = S_\sigma \otimes S_U$ with $S^0_{Prod} = S^0_\sigma \otimes S^0_U$ as initial states, and we have a transition between two states if either corresponding transitions with an equal event type σ_I existed in both the incoming event automaton and the functional unit automaton (we say that the incoming event automaton and the functional unit automaton synchronize on such transitions), or if a transition with an empty incoming event ε existed in the functional unit automaton:

$$\begin{aligned} T_{Prod} \quad = \quad & \{((u,v),\sigma_I,[d_l,d_u],\sigma_O,(u',v'))| \\ & (u,\sigma_I,u') \in T_\sigma \wedge (v,\sigma_I,[d_l,d_u],\sigma_O,v') \in T_U\} \\ & \cup\{((u,v),\sigma_I,[d_l,d_u],\sigma_O,(u,v'))| \\ & (v,\sigma_I,[d_l,d_u],\sigma_O,v') \in T_U \wedge \sigma_I = \varepsilon\} \end{aligned}$$

Since we only keep reachable states as well as synchronized transitions and transitions with empty events, the so-obtained product automaton is often considerably smaller than the ordinary automaton product of the input event automaton and the functional unit automaton.

EXAMPLE 3. *Figure 5 shows on the left side the product automaton F_{Prod} of the event automaton F_σ from Example 1 and the functional unit automaton F_U from Example 2. Note, the product automaton consists of only 5 states, since 7 states of the initial product are not reachable.*

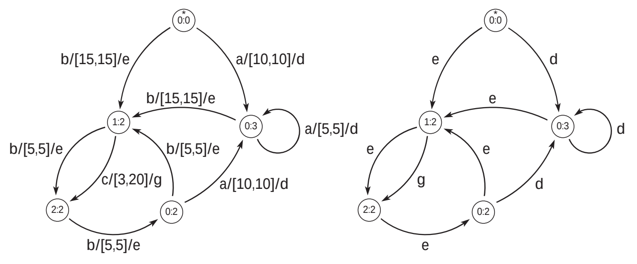

Fig. 5. The product automaton F_{Prod} of the F_σ from Example 1 and F_U from Example 2 (*left*), and the output event automaton $F_\sigma{}'$ from Example 4 (*right*).

A.1 Output Event Automaton

To obtain the output event automaton $F_\sigma{}'$ from the product automaton F_{Prod}, we first relabel the transitions in F_{Prod}:

$$s \xrightarrow{\sigma_I/[d_l,d_u]/\sigma_O} s' \Rightarrow s \xrightarrow{\sigma_O} s'$$

We then need to dispose of all transitions with empty events ε. For this, we apply the following rule to all outgoing transitions of state s', before removing the transition from s to s':

$$s \xrightarrow{\varepsilon} s' \xrightarrow{\sigma_O} s'' \Rightarrow s \xrightarrow{\sigma_O} s''$$

We apply this rule recursively until no transitions with empty events ε exist anymore, which leaves us with the output event automaton $F_\sigma{}'$.

EXAMPLE 4. *To obtain the output event automaton from the product automaton in Example 3, we only need to relabel the transitions with only the output event σ_O. The resulting output event automaton is shown in Fig. 5 on the right. The topological structure of the output event automaton and the product automaton are the same in this example, because there are no transitions with empty events present in the product automaton.*

A.2 Input Workload Curves

The upper and lower input workload curves of a module, denoted by $\gamma^u_I(e)$ and $\gamma^l_I(e)$, bound the maximum and minimum resource demand that any event stream sequence of length e may create in the module.

To compute the upper input workload curve γ^u_I, we start from the product automaton F_{Prod} and first relabel the transitions:

$$s \xrightarrow{\sigma_I/[d_l,d_u]/\sigma_O} s' \Rightarrow s \xrightarrow{\sigma_I/d_u} s'$$

Then we need again to dispose of the empty events ε by applying following rule:

$$s \xrightarrow{\sigma_I/d_{u1}} s' \xrightarrow{\varepsilon/d_{u2}} s'' \Rightarrow s \xrightarrow{\sigma_I/(d_{u1}+d_{u2})} s''$$

We apply this rule recursively until no transitions with empty events ε exist anymore. We then relabel the transitions of this automaton once more:

$$s \xrightarrow{\sigma_I/d_u} s' \Rightarrow s \xrightarrow{d_u} s'$$

In the so obtained automaton, we interpret the resource demand d_u as the weight of a transition. The value of the upper ingoing workload curve $\gamma^u_I(e)$ equals then the weight $w^u(e)$ of the maximum-weight path with length e in this automaton. Techniques to efficiently find maximum-weight paths of any length in a weighted graph are described in [3] and [8].

For the the lower input workload curve γ^l_I, we must follow the same procedure as described above, but at the beginning we retain the lower resource demand d_l. The value of the lower ingoing workload curve $\gamma^l_I(e)$ then equals the weight $w^l(e)$ of the minimum-weight path with length e in the final automaton.

EXAMPLE 5. *To obtain the automaton needed to compute the input workload curves from the product automaton, we again only need to relabel the transitions of the product automaton. This time with the upper or lower resource demand d^u or d^l respectively. The maximum-weight path analysis in this automaton will show that the event sequence* b,a,b,c *imposes the maximum workload on the processing unit. Figure 6 shows the obtained input workload curves.*

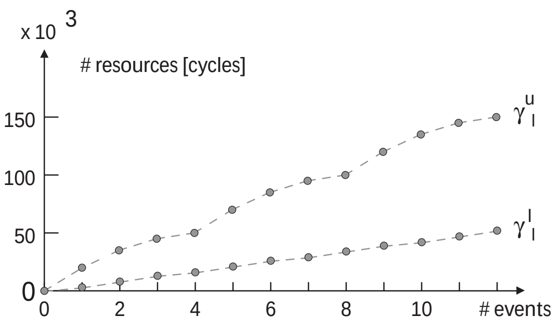

Fig. 6. The input workload curves obtained from the product automaton in Example 3.

A.3 Output Workload Curves

The inverse of the lower and upper output workload curves of a module, denoted by $\gamma^{l^{-1}}_O(r)$ and $\gamma^{u^{-1}}_O(r)$, limit the maximum and minimum number of events that may be produced on the output event stream of a module if any r units of resources are available.

To obtain the output workload curves γ^u_O and γ^l_O, we must follow a similar procedure as the one used to compute the input workload curves. But during the first relabeling, we must retain the output event σ_O instead of the input event σ_I. Further, to dispose of the empty event ε, we must use the same transformation as we used to obtain the output event automaton, where we however again keep the sum of the resource demands as we did for the input workload curves.

B. Workload Transformations (WLT/IWLT)

During the workload transformation WLT, we use the input workload curves γ_I^u and γ_I^l to transform the event based arrival curves $\bar{\alpha}^u$ and $\bar{\alpha}^l$ into resource demand based arrival curves α^u and α^l, and during the inverse workload transformation IWLT on the other side, we use the pseudo-inverse of the output workload curves γ_O^u and γ_O^l to transform the outgoing resource based arrival curves $\alpha^{u'}$ and $\alpha^{l'}$ back into event based arrival curves $\bar{\alpha}^{u'}$ and $\bar{\alpha}^{l'}$:

$$\alpha^l(\Delta) = \gamma_I^l(\bar{\alpha}^l(\Delta)) \qquad \alpha^u(\Delta) = \gamma_I^u(\bar{\alpha}^u(\Delta)) \tag{3}$$

$$\bar{\alpha}^{l'}(\Delta) = {\gamma_O^u}^{-1}(\alpha^{l'}(\Delta)) \qquad \bar{\alpha}^{u'}(\Delta) = {\gamma_O^l}^{-1}(\alpha^{u'}(\Delta)) \tag{4}$$

C. Real-Time Calculus (RTC)

Given resource demand based arrival curves α^u and α^l and the service curves β^u and β^l, the outgoing resource based arrival curves $\alpha^{u\prime}$ and $\alpha^{l'}$ and the remaining service curves $\beta^{u\prime}$ and $\beta^{l'}$ are related by the following formulas [2]:

$$\begin{aligned}\alpha^{l\prime}(\Delta) &= \min\{\inf_{0\le\mu\Delta}\{\sup_{\lambda>0}\{\alpha^l(\mu+\lambda)-\beta^u(\lambda)\} \\ &\qquad +\beta^l(\Delta-\mu)\},\beta^l(\Delta)\} \qquad (5)\\ \alpha^{u\prime}(\Delta) &= \min\{\sup_{\lambda>0}\{\inf_{0\le\mu<\lambda+\Delta}\{\alpha^u(\mu)+\beta^u(\lambda+\Delta-\mu)\} \\ &\qquad -\beta^l(\lambda)\},\beta^u(\Delta)\} \qquad (6)\\ \beta^{l\prime}(\Delta) &= \sup_{0\le\lambda\le\Delta}\{\beta^l(\lambda)-\alpha^u(\lambda)\} \qquad (7)\\ \beta^{u\prime}(\Delta) &= \max\{\inf_{\lambda>\Delta}\{\beta^u(\lambda)-\alpha^l(\lambda)\},0\} \qquad (8)\end{aligned}$$

V. System Composition and Analysis

We may now combine the abstract modules that we defined in the last section into a network. Together with the abstract models of all resources, functional units and incoming event streams, we obtain an abstract performance model of a complete system that can be used for performance analysis. In this complete system, different scheduling policies on a resource can be expressed by the way we link and distribute the abstract resources to the abstract modules [2].

EXAMPLE 6. *Figure 7 shows the abstract performance model of a system, where the events of two incoming event streams* A *and* B *are first processed on a shared* DSP*, before being transmitted to two separate buses* Bus1 *and* Bus2*. The linkage of the* DSP *resource with the system modules corresponds to the usage of a preemptive fixed priority scheduler on the* DSP *(see [2] for more details on scheduling).*

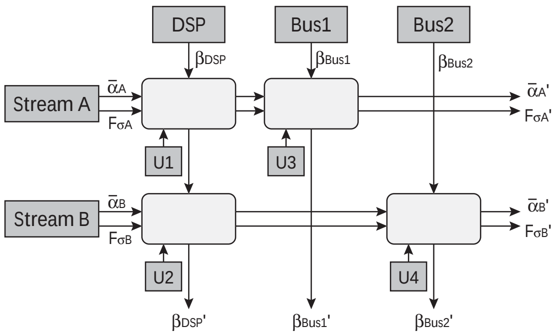

Fig. 7. A system composed of four modules.

We use Real-Time Calculus for performance analysis. In Real-Time Calculus, the maximum delay experienced by an event at a system module, and the maximum number of backlogged resource demand from events of the stream that are waiting to be processed, can be bounded by the following inequalities:

$$delay \le \sup_{t\ge0}\left\{\inf\{\tau\ge0 : \alpha^u(t)\le\beta^l(t+\tau)\}\right\} \tag{9}$$

$$backlog \le \sup_{t\ge0}\{\alpha^u(t)-\beta^l(t)\} \tag{10}$$

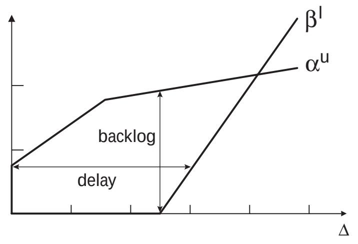

Fig. 8. A graphical representation of delay an backlog obtained from arrival and service curves.

These inequalities are used to compute the delay and backlog of an event stream at every system module and we then sum these results to obtain the delay experienced by an event at the complete system and to compute the needed size of all buffers in the system.

VI. Case Study

In this case study, we demonstrate the advantage of using the presented functional models for modular performance analysis by giving a simple example of a design problem. For the solutions of the problem, we firstly employ traditional worst case analysis, secondly performance analysis using type rate curves to model event streams, and finally we use the new functional models for performance analysis. We compare the results of the three different approaches to show the gain resulting from the application of the new models.

A. Application Scenario

For our example design problem let us consider the functional unit with LRU cache presented in Example 2. The model of this functional unit is shown in Fig. 3. This functional unit runs on an unloaded processor with a fixed processing capacity of $20{\cdot}10^6$ cycles per second and must process the event stream presented in Example 1. The model of this arriving event stream is shown in Fig. 2.

B. The Resource Demand Imposed by the Event Stream

Before we can analyze the application scenario using Real-Time Calculus, we must compute the maximum and minimum resource based arrival curves α^u and α^l. These curves capture the maximum and minimum resource demand imposed by the event stream on the processing unit in any time interval.

In traditional *worst case analysis*, we have no means to exploit functional information of the event stream or the functional unit, and we must assume each arriving event to impose the maximum possible (minimum possible) resource demand (WCET/BCET) any event could impose on the processing unit [11]. In our application, event c has both the maximum and minimum possible resource demand. The resulting arrival curves α^u_{WCET} and α^l_{WCET} are shown in Fig. 9.

Type rate curves [15] model event streams by bounding the number of occurrences of every event type in an event stream sequence of given length. This model allows to capture some possible correlations and dependencies between the different event types on an event stream, but it is not possible to express information on valid event sequences. Furthermore, type rate curves cannot capture any information about system components. In our example, we can therefore not take into account the caching effects of the functional unit. The arrival curves α^u_{TRC} and α^l_{TRC} resulting from an analysis with type rate curves are also shown in Fig. 9.

In difference to the above analysis methods, the new *functional models* allow to capture functional information of both the event stream as well as the functional unit. This allows us to exploit the correlations and dependencies between the different event types on the incoming event stream as well as the caching effects of the functional unit. The resulting arrival curves α^u_{FM} and α^l_{FM} are shown in Fig. 9.

C. Design Problems and Results

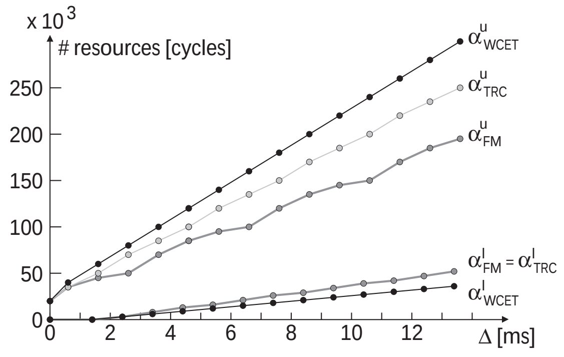

Fig. 9. The continuous upper and lower bounds to the resource demand imposed by the event stream on the processing unit in any time interval, when analyzed using worst case analysis (WCET), type rate curves (TRC) and functional models (FM).

PROBLEM 1. *The maximum delay experienced by an event when processed by our system must not be longer than* $1ms$. *What is the minimum processor frequency required to guarantee the processing of the event stream under this condition? If we build the system with a processor with this minimum processor frequency, how many percent of the processing cycles are guaranteed to remain to process lower priority tasks?*

Using (9) with the different arrival curves shown in Fig. 9, we obtain the minimum lower service curve $\beta^l(\Delta) = f_{min} \cdot \Delta$ and thus the minimum required processor frequencies shown in the second column of Table I. When we then compute the lower outgoing service curve $\beta^{l'}$ using the same arrival curves, we obtain the remaining processing capacities shown in the third and fourth column of Table I.

TABLE I
RESOURCE UTILIZATION ANALYSIS FOR PROBLEM 1.

Analysis Method	f_{min}	Guaranteed Remaining Proc. Capacity	
		f_{min} = 219 MHz	f_{min} = 250 MHz
WCET	250 MHz	—	20 %
Type Rate Curves	219 MHz	23.7 %	33.3 %
Functional Models	219 MHz	42.9 %	50.0 %

From the results in Table I, we see that using functional models for performance analysis, we firstly obtain tighter analysis results for the minimum required processor frequency and moreover we can guarantee more free processing capacity for lower priority tasks than we could with the other methods.

PROBLEM 2. *What is the minimum processor frequency required to guarantee the processing of the arriving event stream if a buffer with a finite size is available and if no requirements on the maximum delay exist? What is then the upper bound to the maximum delay experienced by an event on the event stream?*

We choose the lower service curve $\beta^l(\Delta) = f_{min} \cdot \Delta$ to be in parallel with the tangent to the upper arrival curve α^u with the smallest slope. According to (10), this ensures that there will never be an infinite backlog. We then use (9) to obtain the maximum delay. The analysis results are shown in Table II.

TABLE II
MIN PROC. SPEED AND MAX DELAY ANALYSIS FOR PROBLEM 2.

Analysis Method	f_{min}	d_{max}
WCET	200 MHz	1.4 ms
Type Rate Curves	167 MHz	1.6 ms
Functional Models	125 MHz	2.2 ms

From the above analysis with functional models, we learn that if we can allow slightly longer delays, we can choose a processor with a considerably lower processing speed that is still guaranteed to process the incoming event stream with finite buffers and a bounded maximum delay.

VII. CONCLUSIONS

We presented new abstract models for event streams and system components of embedded hard real-time systems, and we showed how these models can be combined to modules that can be used to abstractly model complete systems for analytic performance analysis. With the presented models we can capture complex functional properties of a system, like caches, resource demand dependencies of different events in an event stream, and arbitrary up- and down-sampling of event streams in a system component. The presented models allow to flexibly choose the level of detail for every system component, such that single system components that seem to be critical may be modeled very detailed, while others may be modeled with less details. We have shown the applicability of the presented models and methods in the context of performance analysis of a system component with a LRU cache, where a considerable improvement in the analysis of worst-case performance results was achieved compared to conventional methods.

VIII. ACKNOWLEDGEMENT

The authors would like to thank Marcel Verhoef and the anonymous reviewers for their valuable comments that helped to improve this paper.

IX. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
[2] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. 6th Design, Automation and Test in Europe (DATE)*, pages 190–195, March 2003.
[3] G. Cohen, D. Dubois, J. P. Quadrat, and M. Voit. A linear-system-theoretic view of discrete-event processes and its use for performance avaluation in manufacturing. *IEEE Transactions on Automatic Control*, AC-30(3), 1985.
[4] R. Cruz. A calculus for network delay. *IEEE Trans. Information Theory*, 37(1):114–141, 1991.
[5] C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *In proc. of RTCSA'99*. IEEE Press, 1999.
[6] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 254–265, June 2001.
[7] M. Jersak, R. Henia, and R. Ernst. Context-aware performance analysis for efficient embedded systems design. In *Proc. 7th Design, Automation and Test in Europe (DATE)*, 2004.
[8] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, (23):309–311, 1978.
[9] J. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. LNCS 2050, Springer Verlag, 2001.
[10] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.
[11] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
[12] A. Maxiaguine, S. Künzli, and L. Thiele. Workload characterization model for tasks with variable execution demand. In *Proc. 7th Design, Automation and Test in Europe (DATE)*, 2004.
[13] K. Richter, M. Jersak, and R. Ernst. A formal approach to mpsoc performance verification. *IEEE Computer*, 36(4):60–67, April 2003.
[14] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.
[15] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative characterization of event streams in analysis of hard real-time applications. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.