A Performance Analysis Framework for Real-Time Systems Sharing Multiple Resources

Shayan Tabatabaei Nikkhah, Marc Geilen, Dip Goswami, and Kees Goossens
Eindhoven University of Technology, the Netherlands
Email: {s.tabatabaei.nikkhah, m.c.w.geilen, d.goswami, k.g.w.goossens}@tue.nl

***Abstract*—Timing properties of applications strongly depend on resources that are allocated to them. Applications often have multiple resource requirements, all of which must be met for them to proceed. Performance analysis of event-based systems has been widely studied in the literature. However, the proposed works consider only one resource requirement for each application task. Additionally, they mainly focus on the rate at which resources serve applications (e.g., power, instructions or bits per second), but another aspect of resources, which is their provided capacity (e.g., energy, memory ranges, FPGA regions), has been ignored. In this work, we propose a mathematical framework to describe the provisioning rate and capacity of various types of resource. Additionally, we consider the simultaneous use of multiple resources. Conservative bounds on response times of events and their backlog are computed. We prove that the bounds are monotone in event arrivals and in required and provided rate and capacity, which enables verification of real-time application performance based on worst-case characterizations. The applicability of our framework is shown in a case study.**

I. Introduction

Mapping applications onto resources is an essential step in design and development of computational platforms. A platform that hosts applications must provide correct functionalities to users. However, besides the functional requirements, users often have non-functional requirements such as performance, security, and reliability demands. The performance of an application, on which we focus in this work, depends on resource shares allocated to that application (e.g., processor cycles, memory space, network bandwidth). Therefore, to realize sound application mappings, services provided by platforms must be sufficient to fulfill application performance requirements.

Services either provided by resources or required by applications, can be expressed at various abstraction levels. In the cloud context, cloud providers describe their provided resources using high abstraction level quantities such as the number of virtual CPUs (vCPUs) and the number of gigabytes of RAM. In the edge computing context, where platforms with limited resources are employed to run tasks, services contain more details. Application deployments using containers (e.g., Docker containers [1]) are common in edge computing frameworks. Compared to services used by cloud providers and users, services for containers are described at lower abstraction levels such as *(quota,period)* which denotes *quota* milliseconds of CPU in every *period*. In the embedded context, abstraction levels can go even lower. For instance, in a platform where processors are scheduled using the TDMA scheme, the CPU service can be described by specifying the exact TDM slots within a TDM frame [2]. An effective modeling framework should allow applications to express their required services at different abstraction levels for all types of resources (e.g., processors, memories, I/O).

This research has received funding from the Electronic Component Systems for European Leadership (ECSEL) Joint Undertaking under grant agreement No 783162 (FitOptiVis project). This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme.

Almost always, applications require multiple resources to execute. The performance analysis methods proposed in the literature either consider only one type of resource (e.g., a computation resource) or perform modular analysis whereby applications are broken into separate tasks each mapped onto one resource. However, in practice, applications often require multiple resources at the same time for their execution. For instance, on battery-powered platforms, besides the computation, communication, and storage resources, applications require energy without which other resources cannot be used.

The contributions of our mathematical framework are:

- a generic resource model whereby different types of resource can be described by its rate and capacity
- a flexible model to describe the required service of applications whereby any requirement pattern can be expressed (i.e., not only constant worst-case requirements)
- modeling simultaneous use of multiple resources.

II. Analysis

Services are required by applications when they are invoked by incoming events (e.g., video frames, DMA transactions, OpenCL kernel calls). A stream of incoming events is modeled by a request function [3], [4] represented by a of monotone function $I : T \rightarrow E$ which shows the total number of incoming events up to a certain time instant (e.g., $\textcircled{1}$ in Fig. 1). We use discrete-time models in this work. Hence, we have $T = \mathbb{N}_0$. Also, we assume that events arrive atomically (i.e., $E = \mathbb{N}_0$). Each application requires a certain service from one or more resources to handle an event. We assume that the required service of applications can be characterized by a set of functions $B_i : P \rightarrow S$ that specify the service that an application requires from resource i when certain application progress is made (e.g., $\textcircled{2}$ in Fig. 1). A service is modeled by a set of pairs $(q, u) \in \mathbb{R}_{\geq 0} \times \mathit{ST}$ which contain the quantity q of service (a non-negative real number, e.g., # memory blocks, # processors in a cluster) and a unit u which shows the type

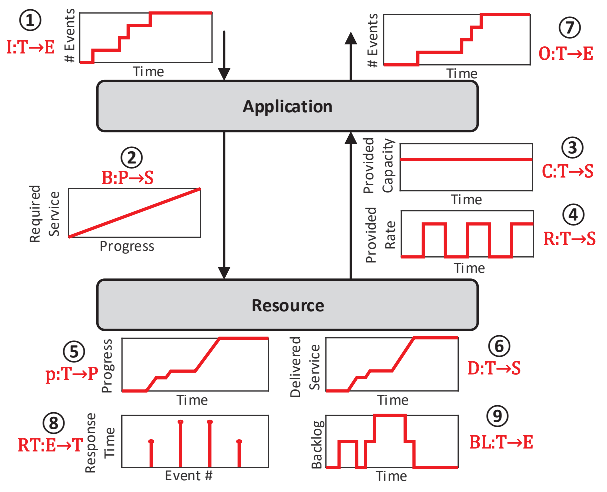


Fig. 1: Single application, single resource analysis

of service (chosen from *ST*, the set of all service types, e.g., cycles, kilobytes, joules, watts, quality level).

The provided service of a resource is modeled by two functions describing the capacity of the resource (i.e., $C_i : T \to S$, see ③ in Fig. 1) and the rate at which the resource serves the requestors (i.e., $R_i : T \to S$, see ④). The capacity of a resource describes the maximum service that the resource can deliver to applications at all the time instants (e.g., battery energy), and the rate describes the maximum service that the resource can deliver to applications at each time instant (e.g., battery power). Resources such as processors and interconnect resources do not have any constraints on the total service they can allocate to applications; however, the service they deliver at each time instant is constrained by their limited bandwidth. In other words, their provided capacity is infinite, but their provided rate is limited. Another class of resources such as memories (space) and FPGA (area) can only accept requests when the total service they deliver to applications at that point has not reached their capacity. Finally, another class of resources such as batteries has limits on both their provided capacity (joules) and rate (watts).

Application progress is a non-negative rational number (i.e., $P = \mathbb{Q}_{\geq 0}$) indicating the number of (fully or partially) processed events. Applications make progress only when they are delivered the services they require from all resources. In other words, all the required resources must provide enough service at the same time, and application progress is limited by a resource whose delivery results in the least progress. Thus, we define $p_i : T \to P$ as a function that specifies the maximum progress of an application if it only requires resource i. Besides being constrained by the availability of resources, application progress is limited by the number of events (i.e., without workload, there is no progress). Accordingly, the progress that an application reaches at time t ($p : T \to P$) is computed as:

$$p(t) = \mathit{min}(I(t), \mathop{\mathit{Min}}_{i \in R} p_i(t)) \tag{1}$$

where R is the set of all resources and $p_i(t)$ depends on the service that the application requires (given by B_i) as well as the service that is provided by resource i at time t. An example of application progress is indicated by ⑤ in Fig. 1. At each time instant, $p_i(t)$ is the progress at which the total delivered service does not exceed the provided capacity, and the service that is delivered since the previous time instant does not exceed the delivery rate:

$$\begin{aligned} p_i(t) = \mathit{Max}\{\rho \geq p(t-1) :& \\ B_i(\rho) - B_i(p(t-1)) \sqsubseteq R_i(t) \ \wedge& \\ \forall p(t-1) \leq \lambda \leq \rho : B_i(\lambda) \sqsubseteq C_i(t)\}.& \end{aligned} \tag{2}$$

Where $A \sqsubseteq A'$ is true when A' contains all the services that A has, and their values are not less than those in A, and the subtraction operator subtracts the values of common services and keeps the uncommon ones intact. Note that since the required capacity is not necessarily monotonic (e.g., required memory can decrease), in Eq. 2, we compare the required to provided capacity for all the progress points from $p(t-1)$ to ρ. For resources offering non-returnable services (e.g., processor cycles) at certain rates, the required service shows the accumulative required service and is non-decreasing. At each time instant, the service delivered to an application by resource i depends on the progress that the application made up to that instant (denoted by $D_i : T \to S$, e.g., ⑥ in Fig. 1). Accordingly, we have $D_i(t) = B_i(p(t))$.

The application performance is given by the output event stream. $O(t)$ denotes the total number of outgoing events up to time t (e.g., ⑦ in Fig. 1) and is derived from the progress function by $O(t) = \lfloor p(t) \rfloor$. From this, we can compute the response time of each event and the backlog at each time instant. To do so, we first define two functions that map the incoming/outgoing event number to the time at which the event arrives/finishes. Arrival/finishing time of events are defined as:

$$A(n) = \mathit{Inf}\{t : I(t) \geq n\}, \quad F(n) = \mathit{Inf}\{t : O(t) \geq n\} \tag{3}$$

where $A(n)$ is the arrival time of the n^{th} event, and $F(n)$ is its finishing time. Subsequently, the (worst-case) response time of events is computed as:

$$\mathit{RT}(n) = F(n) - A(n), \qquad \mathit{WCRT} = \mathop{\mathit{Sup}}_{n \in \mathbb{N}} \mathit{RT}(n) \tag{4}$$

where $\mathit{RT}(n)$ is response time of the n^{th} event (e.g., ⑧ in Fig. 1), and *WCRT* is the worst-case response time of all events. The backlog, which shows the required buffer size (e.g., ⑨), is computed as $\mathit{BL}(t) = I(t) - O(t)$, and the worst-case backlog is its maximum value.

III. Bounds of Application Progress

Our analysis method works when the concrete traces of event streams, required resources, and behavior of resources are known. However, usually, these concrete traces are not known at design time (or even run time), but the bounds of all the traces are given (denoted by l and u for lower and upper bounds). We first show that the application progress is monotone in event arrivals, provided service, and required service.

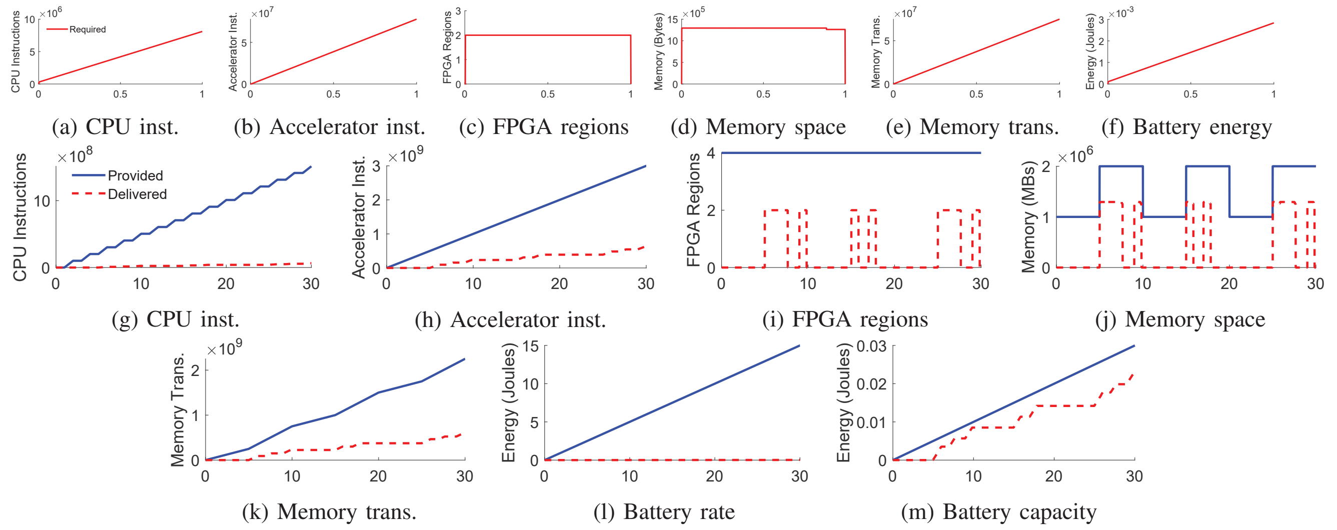


Fig. 2: Required ((a)-(f), plotted against progress), provided ((g)-(m)), and delivered ((g)-(m), plotted against time) services

Using this, we compute bounds on application progress. All the following theorems are proved by mathematical induction on t (they are omitted for lack of space).

Theorem 1. *(Progress is monotone in event arrivals) Let $I(t)$ and $I'(t)$ be two event streams of an application such that for any time instant $t : I(t) \leq I'(t)$, and let $\{(R_i, C_i)\}$ be a set of rates and capacities provided to the application. Then, for the application progress we have $\forall t : p(t) \leq p'(t)$.*

Theorem 2. *(Progress is monotone in provided service) Let (R_i, C_i) and (R'_i, C'_i) be two traces of the service provided by resource i such that for any time instant $t : R_i \sqsubseteq R'_i$ and $C_i \sqsubseteq C'_i$. Let $I(t)$ be an event stream. Then, $\forall t : p(t) \leq p'(t)$.*

Theorem 3. *(Progress is monotone in required service) Let $I(t)$ be an event stream and $\{(R_j, C_j)\}$ a set of services provided to an application. Additionally, let B_i and B'_i be two required service functions of resource i such that at any progress (point) $\rho : B_i(\rho) \sqsubseteq B'_i(\rho)$. Then, $\forall t : p(t) \geq p'(t)$.*

Using these theorems, we can compute bounds of application progress (i.e., p^l and p^u) using the bounds of incoming events, provided services, and/or required services . For this purpose, we replace them in Eq. 1 and 2 with their bounds. Subsequently, using the (monotone) floor function, bounds of the number of outgoing events are computed (e.g., $O^l = \lfloor p^l \rfloor$). We obtain *RT* and *BL* upper bounds using the bounds of incoming and outgoing events. For example, the upper bounds of response times are conservatively computed as $RT^u(n) = F^u(n) - A^l(n)$ where:

$$F^u(n) = Inf\{t : O^l(t) \geq n\}; \quad A^l(n) = Inf\{t : I^u(t) \geq n\}.$$

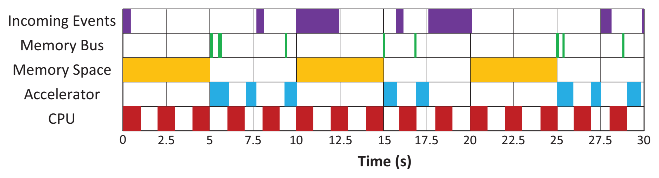
IV. An Illustrative Case Study

In this section, we show the applicability of our proposed framework using an illustrative example. In this example, we consider a battery-powered platform containing a processor (providing 100 MIPS computation power) coupled to a 2 MB memory through a bus with 100 mega transactions per second bandwidth and an FPGA accelerator through a separate bus. The battery harvests energy at 1 mW, has a maximum power of 0.5 W, and is initially out of charge. The platform runs few applications including a heterogeneous implementation of JPEG decoder where *IDCT* and *IQZZ* tasks are offloaded to the accelerator, and other tasks run on the processor. Required services of a single-core JPEG decoder have been characterized using Valgrind [5], and the heterogeneous implementation is characterized by modifying the model of the single-core version. A stream of incoming events for the JPEG application is characterized as $(period, jitter) = (4\ sec, 0.2\ sec)$.

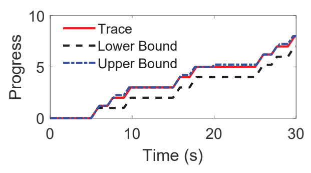
The services required by the JPEG decoder to execute one event are shown in Fig. 2a to 2f. Here, progress is the normalized number of executed CPU and accelerator instructions. The provided and delivered services are depicted in Fig. 2g to 2m. Note that for resources whose provisioning rates are limited, the accumulated provided services are shown (e.g., Fig. 2g). It can be seen in Fig. 2j that due to the usage of other applications, at some time instants, the provided memory capacity is 1 MBs which is not enough to run JPEG decoder. Hence, the application cannot make progress at these time instants, which affects its response times. Consequently, if the memory allocations were static, the JPEG decoder could not be mapped on the platform, unlike our case where applications can share the same memory space.

Having computed the maximum progress that an application can reach if it only requires one resource (i.e., p_i), we can identify the resources which have slowed down the application progress (i.e., bottlenecks). Fig. 3a depicts the bottlenecks over time. We can see the major bottlenecks are the memory space and the processor which are shared among the running applications. However, other resources such as the accelerator have also slowed down the progress at some time intervals.

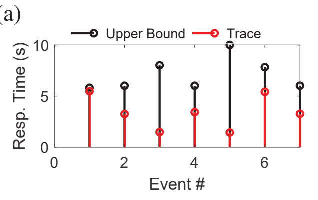
The computed bounds for the application progress and response times of event are illustrated in Fig. 3b and 3c .



(a)



(b)



(c)

Fig. 3: Bottlenecks (a), progress (b), and response times (c)

The actual values for the application trace is also shown for comparison. The bounds of incoming events are computed according to [6]. It can be seen that in all cases the computed bounds are conservative and can be used to analyze the application based on the worst-case characterizations.

V. Related Works

Performance analysis is a crucial step in both designing real-time systems and their run-time adaptations. Hence, works related to performance analysis cover a wide span of research scopes ranging from designing embedded systems to resource management in clouds. In the cloud and fog computing context, the *task offloading* and *Virtual Machine placement* tasks boil down to mapping applications to computational platforms (e.g., edge devices) such that their *Quality of Service* demands (e.g., latency requirements) are met and a cost function is minimized. Numerous solutions are proposed to solve this problem (see [7], [8]) where the common step is comparing the required resources of applications/VMs to resources provided by platforms. The proposed solutions consider simple models (e.g., maximum memory usage, number of virtual CPUs) for resource services, which leads to design for the worst-case and degrades the resource utilization.

In the embedded domain, more detailed models and mathematical approaches have been proposed. Network Calculus (NC) [3] is a mathematical framework that models traffic flows and network components to analyze the performance of message flows by computing bounds of delay and backlogs. Inspired by NC, Real-Time Calculus (RTC) [4], [9] provides an analytical methodology for embedded real-time systems to determine the performance of event-driven tasks running on communication and computation resources. Interface-based design [10] uses RTC to build and analyze complex systems by composing simpler components using their interfaces. RTC is used to check the feasibility of compositions. Many other analytical frameworks are built on top of RTC and interface-based compositional analysis (e.g., [11], [12]), which consider more complicated settings such as probabilistic real-time systems and multi-mode components. However, RTC and its variants only consider computation and communication resources whose models only contain provisioning/consumption rates, and resources with limited capacity are not included in their frameworks. Additionally, for each component, they consider requirements on only one resource, which limits their practicality. Synchronous Data Flow Graphs and its variants are modeling techniques whereby application tasks can be modeled by a set of actors connected by channels [13]. Each actor is mapped to one resource, and based on the provided service of the resource, timing analysis can be performed. Although various types of resources and scheduling policies are considered, the models do not consider multiple resource requirements for each actor (e.g., processor cycles and battery energy). Additionally, the dynamic resource allocation (e.g., dynamic memory) is not modeled.

Considering all the above, we propose a mathematical framework where every type of resource is modeled by its rate and capacity (rather than just rate). Additionally, using the notion of progress, multiple resource requirements for a single application as well as complex requirement patterns can be expressed. The monotonicity of our approach allows us to use worst-case characterizations to compute bounds on application progress (i.e., real-time behavior).

VI. Conclusion

Our proposed mathematical framework extends the existing performance analysis approaches by offering a generic model for every type of resource and considering multiple resource requirements at the same time. The applicability of our proposed framework is illustrated in the JPEG decoder case study, which shows how our models can improve resource utilization.

References

[1] D. Merkel, “Docker: lightweight Linux containers for consistent development and deployment,” *Linux Journal*, vol. 2014, no. 239, 2014.
[2] S. Sinha *et al.*, “Composable and predictable dynamic loading for time-critical partitioned systems on multiprocessor architectures,” *Microprocessors and Microsystems*, vol. 39, no. 8, 2015.
[3] J.-Y. Le Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*. Springer Science & Business Media, 2001, vol. 2050.
[4] L. Thiele, S. Chakraborty, and M. Naedele, “Real-time calculus for scheduling hard real-time systems,” in *ISCAS*, 2000.
[5] N. Nethercote, R. Walsh, and J. Fitzhardinge, “Building workload characterization tools with valgrind,” in *ISWC*, 2006.
[6] E. Wandeler, A. Maxiaguine, and L. Thiele, “On the use of greedy shapers in real-time embedded systems,” *TECS*, vol. 11, no. 1, 2012.
[7] B. Jennings and R. Stadler, “Resource management in clouds: Survey and research challenges,” *Journal of Network and Systems Management*, vol. 23, no. 3, 2015.
[8] C.-H. Hong and B. Varghese, “Resource management in fog/edge computing: A survey,” *arXiv preprint arXiv:1810.00305*, 2018.
[9] S. Chakraborty, S. Künzli, and L. Thiele, “A general framework for analysing system properties in platform-based embedded system designs.” in *DATE*, 2003.
[10] E. Wandeler and L. Thiele, “Interface-based design of real-time systems with hierarchical scheduling,” in *RTAS*, 2006.
[11] L. Santinelli and L. Cucu-Grosjean, “A probabilistic calculus for probabilistic real-time systems,” *TECS*, vol. 14, no. 3, 2015.
[12] L. T. Phan, I. Lee, and O. Sokolsky, “Compositional analysis of multi-mode systems,” in *ECRTS*, 2010.
[13] S. Sriram and S. S. Bhattacharyya, *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2018.