End-to-End Analysis of Event Chains under the QNX Adaptive Partitioning Scheduler

Dakshina Dasari[1] Matthias Becker[2] Daniel Casini[3] Tobias Blaß[4]
[1]*Robert Bosch GmbH, Germany*
[2]*KTH Royal Institute of Technology, Stockholm, Sweden*
[3]*TeCIP Institute and Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy*
[4]*Robert Bosch GmbH and Saarland University, Saarland Informatics Campus (SIC), Germany*

***Abstract*—Modern autonomous cars run classic AUTOSAR applications alongside advanced driving assistance systems on a single-vehicle computer. Ensuring safety and predictability in such a complex system is challenging and requires temporal isolation between the various components. A promising solution is the POSIX-compliant QNX operating system: it meets the automotive standards for functional safety at the highest level (ISO 26262 ASIL-D) and provides temporal isolation through the *Adaptive Partitioning Scheduler (APS)*, a resource reservation algorithm that guarantees processor bandwidth to groups of threads. These guarantees make it an ideal platform for composing diverse and complex applications on centralized vehicle computers. However, so far, there is no precise description or analysis of the APS reservation mechanism in real-time literature. In this paper, we provide the first description of the behavior of the APS from a real-time point of view and validate the results by running experiments on a real QNX platform. Based on the derived scheduler rules, we develop a response-time analysis to bound the end-to-end latency of event chains under APS. Finally, we evaluate different design strategies on a case study based on a real autonomous construction vehicle.**

I. Introduction

Automotive E/E architecture is witnessing a major disruption on multiple fronts, driven by the requirements of emerging automated driving applications and the aim to realize a truly software-defined vehicle. On the hardware front, there is a shift towards centralized architectures, marked by the integration of previously distributed functions onto powerful microprocessor platforms; on the software front, there is a shift towards commodity POSIX-based operating systems (OS). Classical AUTOSAR-based OS'es, which are designed for static workloads with fixed interactions, are poorly suited to the dynamically arriving, data-driven workloads of emerging autonomous applications and do not cope well with newer use-cases such as modular updates over the product lifetime. As a consequence, POSIX-based operating systems like QNX and Linux have emerged as promising alternatives. QNX, in particular, is widely preferred as the base operating system by many automotive OEMs, as it is ISO-26262 certified at the highest level of assurance (ASIL-D).

The move towards a centralized architecture requires system designers to integrate multiple independent applications with different timing requirements on the same centralized platform. They need to ensure temporal isolation among the executing applications while still utilizing the system efficiently. An effective approach to realize this are OS mechanisms that provide guarantees on the provision of system resources like CPU-time [1, 2] or memory bandwidth [3, 4]. This way, designers can ensure that each executing application receives enough resources to meet its timing requirements.

To this end, the *Adaptive Partitioning Scheduler* (APS) offered by the QNX OS [5] is a promising candidate, and the subject of this work. APS is a resource reservation mechanism, where threads are grouped into virtual containers called *partitions*. Each partition is guaranteed a certain amount of processing bandwidth in a given interval of time, irrespective of what is executed on other partitions, which ensures temporal isolation among co-running applications. In addition, QNX APS optionally redistributes spare computation time if partitions do not use their entire budgets, thereby avoiding the problems associated with static allocations and enabling efficient system utilization. Compared to the reservation-based scheduler in Linux, the `SCHED_DEADLINE` scheduling class [6], it supports more flexible deployments in that it allows multiple threads to share the same reservation budget. This flexibility is important in various practical applications, as recently noted in the context of ROS 2 and DDS [7].

Despite the benefits provided by the QNX APS, its adoption in practical systems is limited. A likely reason is that neither a formal description of its behavior, nor a response-time analysis, nor prior case studies regarding the practical implementation issues on an actual platform are available. One challenge towards such a description and analysis is that although APS builds on well-established real-time system concepts, the exact reservation mechanism is different from every other reservation-based scheduler studied in the past. First, unlike reservation servers [8], APS partitions are not associated with a single priority but only act as a "container" for the underlying threads, each of which has its own priority. Second, they do not periodically recharge their current budget to the nominal value but rely on a sliding accounting window instead, which gradually replenishes the budget over time. Third, it implements QNX-specific budget reclamation policies and mechanisms to handle budget overruns.

In this work, we study the APS scheduler from a timing perspective and consider the problem of bounding the end-

to-end latency of automotive applications. An automotive application is typically realized as an event-chain consisting of a coordinated chain of functions, which exchange information to realize a given behavior (analogous to a graph, where the nodes represent the functions and the edges represent the messages passed between them). In these applications, event chains are associated with an end-to-end deadline specifying the latest acceptable time by which an input at the first task of the chain leads to an output at the last task in the chain.

To guarantee such end-to-end deadlines, the resource reservation mechanism implemented by APS can provide essential benefits. Event chains can be decomposed and deployed to APS partitions, which provide a guaranteed supply time *irrespective* of the workload running in other partitions. This is a key mechanism to abstract from the interference that may arise in a complex system and enables *compositional analysis* [2, 9]. However, such deployment does not come for free. The system designer needs to answer several non-trivial questions to guarantee that each application event chain meets its end-to-end deadlines: how much processing time does each partition need? How to set priorities for threads? And how to map threads to cores in a multi-core platform?

Contributions. Answering these questions needs an understanding of the real-time behavior of event chains running on APS. This calls for an analysis-driven solution, which can be derived only after careful modeling of the APS. In this paper, we make the following contributions. We formalize the APS through a set of rules. Each rule has been validated with a corresponding experiment on a Raspberry Pi running QNX 7.1. By building on the proposed model, we propose a response-time analysis for event chains running on the QNX APS. Furthermore, we validate our analysis on a real platform with a case study based on an autonomous construction vehicle, which we use as a testbed to evaluate different design strategies. Finally, we uncover some practical issues when using APS for hosting event chains: we discuss quantization effects in the time accounting of QNX and discuss the most suitable interprocess mechanism with APS to realize chains.

II. Overview of the QNX APS

The Adaptive Partitioning Scheduler provided by QNX allows implementing *resource reservation* for groups of threads or processes, which are grouped within virtual containers called *partitions*. Each partition can be configured with a percentage of the overall processing capacity. This is specified by configuring a per-partition *budget*, which determines the amount of processing time that can be used by a partition in a time window. APS throttles the CPU usage of each partition by measuring its CPU usage in a configurable sliding window, which is set to 100 ms by default, and it is common to all partitions in the system. The APS prevents a thread that exceeded its budget in the time window from executing whenever there are other ready-to-run threads in other partitions with available budget. The partition's budget is gradually restored when enough time has elapsed.

APS performs time accounting for the budgets whenever a scheduling event occur: examples of scheduling events are the timer tick, a thread termination, or a message arrival. Every time it executes, APS selects the highest-priority thread whose partition has available budget [10]. It thereby combines the principles of fixed-priority preemptive scheduling and budget enforcements. APS allows the configuration of up to 32 partitions.

APS also provides the possibility to setup a *critical budget*, which specifies the amount of time a partition is allowed to use when its regular budget is depleted. The critical budget may be configured to provide additional time to threads, but it is supposed to be used rarely and only to serve highly latency-sensitive applications. Furthermore, it provides a budget reclamation mechanism (called the *idle-time mode*) to improve the average-case performance whenever there are no other partitions with ready threads that can execute on the cores. The idle-time mode allows partitions that are out of budget to execute their ready threads, using two different policies to select which ready thread to run: the first is by its static priority (i.e., select the partition having the highest priority thread), and the second one is by using a metric that considers the ratio of the partitions' budgets. APS also allows specifying a maximum budget that limits the amount of time a partition can overrun its normal budget: this allows to the application designer to disable the idle-time mode. QNX also implements other scheduling policies: in this work, we assume them to be disabled.

III. Modeling and Validation

The purpose of this section is twofold. First, we introduce the system model considered in this paper, both in terms of workloads and for the QNX APS behavior. Then, we corroborate our assumptions by reporting the results of some validation experiments we performed on a real embedded system running QNX.

A. System Model

The system considered in this paper is composed of a set $\mathcal{C} = \{c_1, \ldots, c_m\}$ of homogeneous processor cores running the APS scheduler of QNX.

Workload Model. The processing platform executes a set of applications composed of multiple sequential computational activities characterized by cause-effect dependencies and that need to respond within end-to-end deadlines. Computational activities are modeled by a set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ of threads, each of which releases a potentially infinite sequence of instances. Each thread $\tau_i \in \mathcal{T}$ is characterized by a worst-case execution time (WCET) e_i, a globally unique priority π_i, and a core affinity. We assume that threads are independent and do not share resources. In principle, QNX allows arbitrary core affinities; this paper focuses on partitioned scheduling, i.e., each thread has an affinity to only one core, as this setting is particularly suitable to foster predictability [11, 12]. A thread is said to be ready when it is eligible to be executed, and it is said to be pending from its release to when it completes.

The various threads are co-related by cause-effect dependencies, modeled by means of a direct acyclic graph (DAG) $\mathcal{D} = (\mathcal{T}, \mathcal{E})$, where threads represent vertices in the graph and the set $\mathcal{E} \subseteq \mathcal{T} \times \mathcal{T}$ of edges encodes communication relations among them. Each edge $(\tau_p, \tau_c) \in \mathcal{E}$ represents a producer-consumer relation where τ_c consumes the data produced by τ_p and characterizes a precedence constraint between τ_p and τ_c. This means that an instance of τ_c is allowed to be run only when the corresponding instance of τ_p is completed.

Threads may communicate either via shared memory or via the network (e.g., using the QNX Neutrino native networking [13]). The model thus also supports networks of distributed threads, where each computing node runs APS. Each edge $(\tau_p, \tau_c) \in \mathcal{E}$ is therefore characterized by a message-dependent communication delay $\lambda_{p,c}$, which bounds the amount of time required for data produced by τ_p to be available for τ_c. We assume the communication delay to be negligible, i.e., $\lambda_{p,c} = 0$, when the two threads belong to the same APS partition. We consider a discrete model of time, where each time unit is an integer multiple of some basic units (e.g., a processor cycle).

Event Chains. Some of the threads in the graph $\mathcal{D}$ have no incoming or outgoing edges: these threads are said to be *source* and *sink* threads, respectively. Each of source thread gives rise to an event chain γ_x, i.e., a path in the graph.

The set of all event chains is denoted as $\Gamma = \{\gamma_1, \dots, \gamma_a\}$. Each thread is characterized by an event arrival curve $\eta_i(\Delta)$, which bounds the number of release events in any time window of length Δ. Arrival curves are externally provided for source threads; otherwise, they need to be derived whenever needed for other threads, as extensively discussed later in Section IV. Each chain γ_x is characterized by an end-to-end deadline D_x.

Modeling the APS Scheduler. We consider an APS scheduler composed of a set of partitions $\mathcal{P} = \{P_1 \dots P_s\}$. In APS, all partitions share a common accounting window with length W (typically set to 100 ms [5]). While APS is a flexible scheduler that gives a lot of freedom to applications developers, we restrict our analysis to a specific configuration that we deem particularly suitable to foster predictability. Like threads, each partition is logically assigned to only one core (i.e., all threads that are assigned to the partition are assigned to the same core). Each core $c_j \in \mathcal{C}$ can host multiple partitions, which are denoted with the set $\mathcal{P}_j$.

Each partition P_k is associated with a nominal budget of B_k time units. At any point in time t, P_k is also characterized by a *current* budget $b_k(t)$. A partition P_k is ready when at least one of its thread is *ready*, and it is running when one of its thread is *running*. The set of all threads assigned to P_k is referred to as $\mathcal{T}_k$. We assume the critical budget of each partition to be set to 0. An event chain $\gamma_i \in \Gamma$ may span multiple partitions.

The APS scheduler keeps track of any scheduling event that may cause a thread to start or stop running (e.g., a message send or receive). Furthermore, it keeps track of the last absolute time t_l at which the timer interrupt executed or a thread started or stopped running. If none of these events previously occurred, $t_l = 0$.

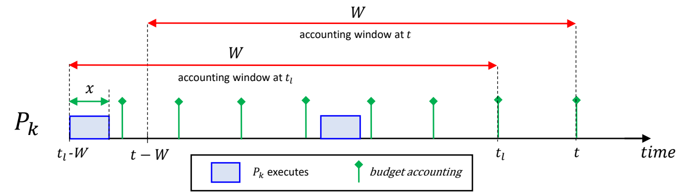


Fig. 1: Budget increment over a sliding window.

We consider two possible settings for the idle-time mode (i.e., budget reclamation): budget reclamation is either disabled or distributes idle time in priority order. The former is configured by setting the idle-time policy to `SCHED_APS_SCHEDPOL_LIMIT_CPU_USAGE` and setting the maximum budget `max_budget_percent` equal to the nominal budget B_k. The latter is configured by setting the idle-time policy to `SCHED_APS_SCHEDPOL_DEFAULT`.

In the following, we describe the behavior of the APS scheduler with a set of rules, considering an arbitrary core (partitioned scheduling) and a partition P_k. The rules are listed in the second column of Table I.

The described behavior has been derived by studying and interpreting the QNX documentation. Manuals are written in natural language: hence, they are sometimes ambiguous. Furthermore, since the QNX source code is not publicly available (*closed source*), the provided descriptions cannot be cross-verified by inspecting the source code. We therefore performed a set of validation experiments – described in the third column of Table I – to corroborate our findings with empirical evidence. In addition to the rules reported in the Table I, we also assume for the analysis we present next:

R8 When the budget of a partition is depleted, the partition (i.e., the thread it is running) is *immediately* descheduled.

Whether this assumption is fulfilled by a QNX system depends on the timer resolution, which may produce small fluctuations depending on the fact that the budget accounting is done both periodically, i.e., in correspondence of the system tick interrupt-service routine (ISR, with period $T_{\text{ISR}} < W$) and sporadically, i.e., in correspondence of system calls. This practical aspect is extensively discussed in Section III-C.

B. APS Scheduler Validation

We validated the APS scheduler rules through a set of experiments on a Raspberry Pi 4B with 4 CPUs and 4GB RAM using the QNX 7.1 Software Development Platform (SDP).

Four different settings are executed on the platform to validate the APS scheduler rules. For each of the settings, the execution trace is obtained using the QNX event tracing facility. The APS partition statistics are logged with a granularity of 1 ms. The tracing tool provided by QNX reports the cumulative time that tasks of a respective partition executed within the last scheduling window $[t - W, t)$. In the rest of the section, we therefore use the term *runtime* to refer to this quantity. In the following charts and tables, the budget

TABLE I: APS Scheduler Behavior and Validation.

	Rule	Validation
R1	At the system startup, $b_k(0) = B_k$. That is, the current budget of each partition P_k at $t = 0$, is initialized to B_k.	Consider the APS traces in Fig. 2. The runtime in the last window is 0 at the beginning of the execution and therefore the maximum budget is available to the partitions.
R2	When a timer interrupt or a scheduling event occurs at time t, and P_k is running at t, then, $b_k(t)$ is decremented by $t - t_l \leq T_{\text{ISR}} < W$.	In the trace of Setting A (Fig. 2a), as threads execute, the corresponding runtime increases (and hence the budget decreases) linearly over time.
R3	When a timer interrupt or a scheduling event occurs at time t and $t_l \geq W$, then the accounting window advances by $t - t_l$ units (Figure 1). The budget at time t is then given by $b_k(t) = \max(B_k, b_k(t_l) + x)$, where x is the amount of time any thread of P_k executed in $[t_l - W, t - W)$. Otherwise, if $t_l < W$, there is no budget increase. This rule is applied after **R2**.	In the trace of Setting A (Fig. 2a), τ_2 has an overall execution time of $150\,\text{ms}$. As it executes in $[20, 100]$, the runtime of P_2 increases by $x = 80$ (overall) and by time $t = 100$, P_2 has consumed its entire budget of $80\,\text{ms}$. Then the budget starts recharging at $t = 120$, but τ_2 also starts executing and consuming the budget (due to **R2**), and therefore the overall runtime (and budget) remains constant. At time $t = 190$, τ_2 has executed for $150\,\text{ms}$ and terminates. So at time $t = 190$, we observe that the runtime starts decreasing, i.e., the budget starts increasing since thread τ_2 has terminated and is no more consuming the budget.
R4	When a timer interrupt occurs at time $t = kT_{\text{ISR}}$ with $k > 0$, and for each scheduling event, if after applying R2 and R3, the partition depletes its budget, i.e., $b_k(t) \leq 0$, and there exist at least one other partition demanding processor time, the currently running thread is descheduled.	In Setting C (Fig. 2c), budget reclaiming is enabled and τ_3 is added to partition P_2. Because of this, τ_1 in P_1 cannot consume more than its allowed budget during idle times as it was possible in Setting A without the additional task.
R5	If budget reclaiming is disabled and $b_k(t) \leq 0$, the partition is descheduled, irrespective of the presence of other ready partitions.	In Setting B (Fig. 2b) budget reclaiming is disabled. A ready thread therefore cannot run if its partition is out of budget, even if idle time is available. This is the case at $t = 190\,\text{ms}$ where τ_2 finishes executing its first job. Partition 1 is out of budget ($20\,\text{ms}$ of execution every $100\,\text{ms}$) but has a ready task. The CPU stays idle.
R6	If budget reclaiming is enabled, let $\mathcal{P}' \subseteq \mathcal{P}_j$ the set of partitions such that for all $P_k \in \mathcal{P}' \implies b_k(t) \leq 0$, for each core $c_j \in \mathcal{C}$. If $\mathcal{P}'$ is not empty and there are no other partitions on the same core with ready threads and positive budget, the highest-priority thread in $P_k \in \mathcal{P}'$ runs.	In Setting D (Fig. 2d), the budget of partition P_2 was reduced compared to Setting C. At $t = 40\,\text{ms}$, both partitions are out of their nominal budget and the threads are scheduled in priority order.
R7	If a scheduling event occurs at time t, then the scheduler runs the highest-priority thread that is ready *and* whose corresponding partition P_k has budget $b_k(t) > 0$ (if any).	Based on the performed experiments, a preemptive priority-based scheduling has been observed.

is reported per core if not otherwise stated. This means, a partition on a single core can have at most a budget of 100%. Internally, QNX keeps track of the budget on a global level, where 100% budget refers to the budget allocated to all cores. Hence, on a 4 core platform the maximum budget allocated to a core would be 25%. Regarding priorities, higher numerical value corresponds to a higher priority. The top part of Figure 2 reports the execution trace of the executing tasks and the partition runtime in the four settings. The bottom part of the figure shows the runtime for two APS partitions, P_1 (in blue) and P_2 (in red).

The first setting, referred to as `Setting A` in the following, executes two tasks (see Table II) on the same core and assigns them to two different partitions P_1 and P_2. The configuration of the two partitions is shown in Table III. Both tasks are mapped to the same core and there are no other tasks assigned to the two partitions. P_1 has a budget of 20% and P_2 has a budget of 80%. Note that τ_1 is thus not schedulable with the nominal partition budget alone. Idle time is redistributed in priority order.

`Setting B` is identical to the previous one but disables budget reclamation. `Setting C` extends on `Setting A` by

adding an additional task τ_3 to partition P_2 on core 3. Idle time is again redistributed in priority order. The task τ_3 consumes the remaining budget of partition P_2 after each job of τ_2 completes. The higher priority task τ_1 in P_1 is not allowed to execute in this case as idle time is no longer available.

`Setting D` utilizes two partitions with 20% budget each (Table VI). Three tasks are mapped to the two partitions, all on the same core. Idle time is redistributed in priority order. In this setting, neither partition has enough budget to complete its workload and thus all tasks have to rely on idle-time which is distributed based on the thread's priority.

TABLE II: Threads for Setting A-B

	WCET [ms]	Period [ms]	Priority	Core	Partition
τ_1	50	200	255	3	P_1
τ_2	150	200	254	3	P_2

TABLE III: Configured Partitions for Settings A-B-C

	Window [ms]	Budget Global	Budget Per Core
P_1	100	5%	20%
P_2	100	20%	80%

TABLE IV: Threads for Setting C

	WCET [ms]	Period [ms]	Priority	Core	Partition
τ_1	50	200	255	3	P_1
τ_2	150	200	254	3	P_2
τ_3	50	600	253	3	P_2

TABLE V: Threads for Setting D

	WCET [ms]	Period [ms]	Priority	Core	Partition
τ_1	50	200	255	3	P_1
τ_2	100	200	254	3	P_2
τ_3	50	600	253	3	P_2

TABLE VI: Configured Partitions Setting D

	Window [ms]	Budget Global	Budget Per Core
P_1	100	5%	20%
P_2	100	20%	20%

It may appear that in Figure 2a, τ_2 executes for a very small amount of time (apparently as the same time as τ_1) shortly after 200ms. The observed effect is caused by the way periodic threads have been implemented. In Figure 2a, this happens between the completion of the first and start of the second job of τ_1. The thread performs a system call to suspend τ_1 until the release of τ_1's second job. This duration is very short, in this case, 60 us. Due to plotting effects and minimum line width, this looks larger in the resulting plot.

C. Quantization Errors in Budget Accounting

In QNX, time management is performed based on ticks [14]. The tick size is assigned depending on the CPU clock frequency, and it is usually set to $T_{\text{ISR}} = 1\,\text{ms}$ in most platforms. This value has been also used for the validation in Table I. The tick size is the basis of all time-related functions of the OS and therefore defines the granularity of task parameters that the OS can realize.

As discussed in Section III-A, APS performs the budget accounting upon each timer interrupt or scheduling event. A side-effect of this budget accounting method is shown in Figure 3. In the considered setting, thread τ_1 is created at time $t = 1.5$ ms, in a partition with a full budget of 4 ms. The system tick ISR runs every millisecond, i.e., at $t = 0, 1, 2, \ldots$ ms. Then τ_1 starts running, and since no other system call is triggered, it is descheduled only at time $t = 6$ ms by the tick ISR. As a result, the corresponding partition exceeded its budget by $0.5\,\text{ms}$, consuming processor time that is reserved for other partitions. Budget quantization issues occur in other well-established operating systems like the Linux `SCHED_DEADLINE` scheduler [6]. Although `SCHED_DEADLINE` provides the `HRTICK_DL` feature to overcome budget quantization overruns (i.e., by using a one-shot timer), it has this feature disabled by default as it may introduce additional overhead [15]. Indeed, these effects are commonly interpreted as operating systems overhead and pragmatically accounted for by inflating the timing parameters as for other overheads and variances from the theoretical model.

Different from Linux, in QNX the budget accounting always happens in the timer interrupt handler, even in the so-called *tickless mode* [14], which is only intended for power-saving purposes. Therefore, to the best of our knowledge, no mechanism is currently implemented to guarantee a fine-grained budget accounting.

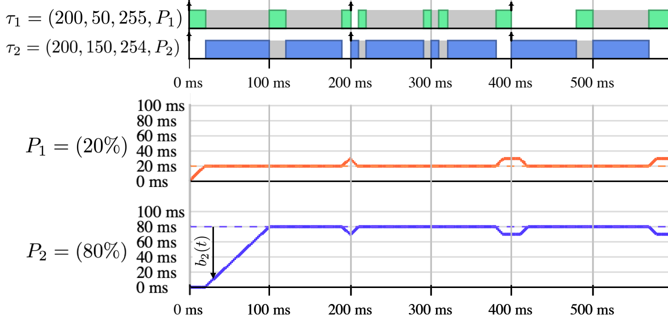
For the rare cases where this can matter, it is possible to mitigate the problem by decreasing the period of the timer interrupts (T_{ISR}) through the `ClockPeriod()` system call. To this end, we performed the same experiment with a higher tick resolution, i.e., with $T_{\text{ISR}} \in \{250, 500\}\,\mu s$ (recall that previous experiments used the standard tick resolution of 1 ms). In both cases, the problem has been solved, and the budget has been correctly accounted for without overruns, as shown in Figure 4. It is worth noting that the size of the APS window W is expressed in *ticks*. Therefore, the window size W needs to be adjusted after any change to the tick resolution T_{ISR}.

However, decreasing the timer period comes at the cost of increased overheads. The application designer should therefore set T_{ISR} to the largest possible period that still provides sufficiently accurate budget accounting for the given application.

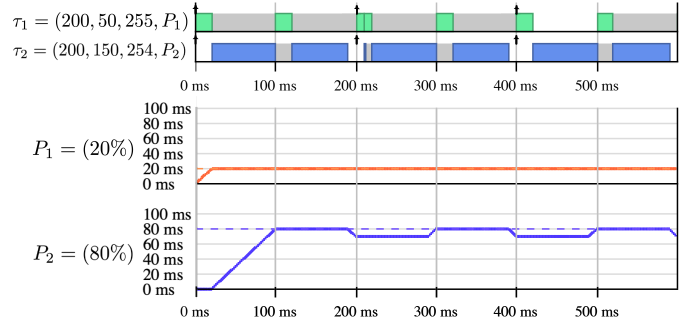
D. Implementing Event Chains in QNX

QNX provides several communication methods that can potentially be used to implement the event-chain model: *channels*, *pulses*, and traditional synchronization primitives like *semaphores*.

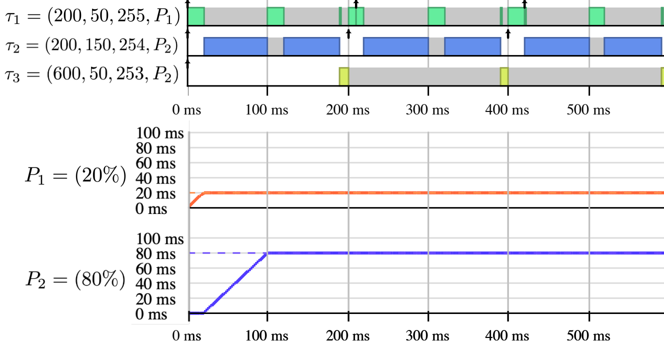
QNX channels are designed for the client/server paradigm. They therefore implement synchronous communication, where the client thread is blocked until the server thread sends a reply [16]. They are therefore a poor fit for the precedence constraints in the timing model, which require non-blocking communication.



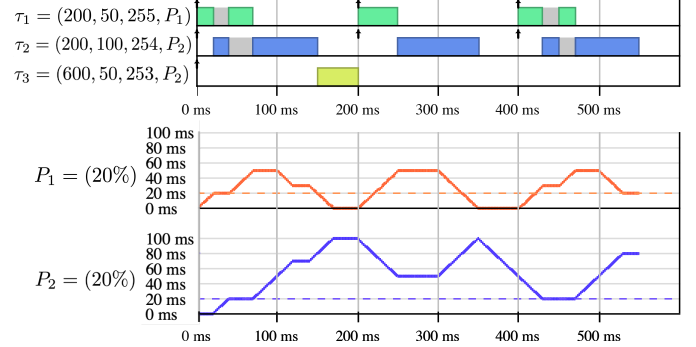
(a) Setting A (With Budget Reclamation), validates R2, R3



(b) Setting B (Without Budget Reclamation), validates R5



(c) Setting C (With Budget Reclamation), validates R4



(d) Setting D (With Budget Reclamation), validates R6

Fig. 2: Execution trace and partition budget usage of the for partitions P_1 (blue) and P_2 (red) in the first $600\,\mathrm{ms}$ of execution. The task parameters are in the following order: period, WCET, priority and assigned partition. Each partition is described by its budget B_k, and $W = 100\,\mathrm{ms}$ in all cases. The current budget $b_k(t)$ is illustrated in Setting A.

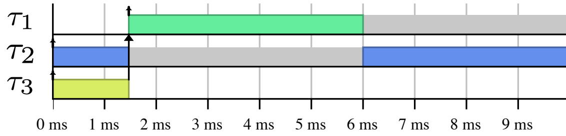


Fig. 3: Budget overrun of 0.5 ms due to a 1 ms resolution of the system timer tick.

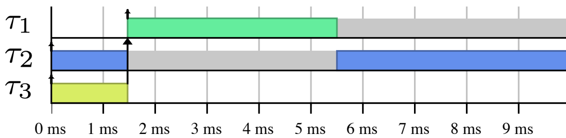


Fig. 4: Correct budget accounting with a 500 μs timer resolution.

As a more light-weight communication method, QNX provides so-called *pulses* as a way to transmit small messages of up to 40 bits in a non-blocking way. However, pulses are treated in a special way under APS scheduling: all pulse message handlers of a process run in the same partition, the so-called *pulse processing partition* [5]. This makes it impossible to isolate threads of the same process from each other.

We therefore decided to avoid QNX communication channels and implement event-chains through traditional semaphores, with one semaphore for each link of an event-chain. Semaphores are lightweight, and do not affect the APS scheduler. Data between threads is exchanged via shared memory and semaphores are used to notify consumer threads about new data items.

IV. Analyzing End-to-End Chains under APS

This section discusses the analysis framework adopted in this paper, which is inspired by Compositional Performance Analysis (CPA) [9]. In particular, we consider the problem of bounding the *worst-case response times* of the event chains (corresponding to the end-to-end latencies) of each application in the system under APS scheduling. The *worst-case response-time* (WCRT) of an arbitrary chain $\gamma_x = (\tau_a, \ldots, \tau_z) \in \Gamma$ is the longest possible time span elapsed from when an instance of its first thread τ_a is released, to when the corresponding instance of its last thread τ_z completes. Note that each thread triggers at most one instance of each of its successor threads. Moreover, a chain could also consist of a single thread with $|\gamma_x| = 1$. We denote with the symbol R_x a bound on the WCRT of γ_x.

To bound the WCRT, we consider each application chain to be composed of a connected sequence of *subchains*, where each subchain $\gamma_{k,h} = (\tau_x, \ldots, \tau_y)$ is a sequence of connected threads, all of which are allocated to the same partition P_k. For such subchains, a WCRT bound is obtained with the results reported next in Section IV-A. However, a complete processing

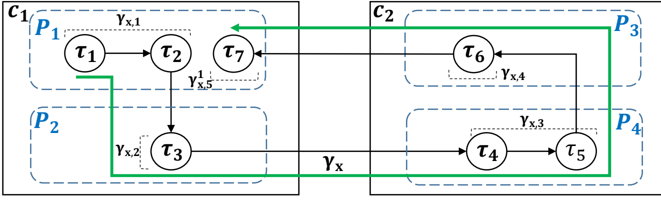


Fig. 5: A chain spanning multiple cores and partitions, and how it is divided into subchains.

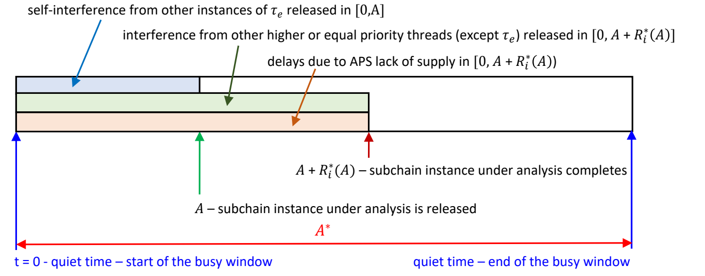


Fig. 6: Timeline showing relevant time instants and intervals for Lemma 1.

chain may span multiple partitions, and may therefore be composed of multiple subchains. The WCRT of the whole chain is thus computed by leveraging the arrival-curve propagation provided by CPA: arrival curves for non-source subchains are derived from the arrival curves of previous subchains and their response-time jitter.

Figure 5 provides an example of the considered setting, where a chain γ_x is divided into five subchains $\gamma_{x,1} = (\tau_1, \tau_2)$, $\gamma_{x,2} = (\tau_3)$, $\gamma_{x,3} = (\tau_4, \tau_5)$, $\gamma_{x,5} = (\tau_7)$, each one consisting of a connected sequence of threads allocated into the same partition P_k in a multicore platform with 2 cores.

The arrival curve of a subchain corresponds to the arrival curve of the first thread in the subchain. Recall that $\lambda_{p,c}$ refers to the communication delay between the producer τ_p and consumer, τ_c. The arrival curve for the source subchain, i.e., $\gamma_{x,1}$, $\eta_1(\Delta)$ is provided externally, while for successor subchains is computed as in [9, 17]: $\eta_3(\Delta) = \eta_1(\Delta + R_{x,1} + \lambda_{2,3})$, $\eta_4(\Delta) = \eta_3(\Delta + R_{x,2} + \lambda_{3,4})$, $\eta_6(\Delta) = \eta_4(\Delta + R_{x,3} + \lambda_{5,6})$, and $\eta_7(\Delta) = \eta_6(\Delta + R_{x,4} + \lambda_{6,7})$. Given the individual response-time bounds for each subchain, the overall end-to-end delay of γ_x is bounded with the sum of response times and communication delays, i.e., $R_x = R_{x,1} + \lambda_{2,3} + R_{x,2} + \lambda_{3,4} + R_{x,3} + \lambda_{5,6} + R_{x,4} + \lambda_{6,7} + R_{x,5}$.

A. Response-Time Analysis

Next, we discuss how to compute the response time for an individual subchain under the assumption that a supply-bound function $sbf_k(\Delta)$, which lower-bounds the minimum service provided by a partition P_k, is known. Then, we show later in Section IV-B how to derive such a function by leveraging our formalization of the APS scheduler.

We define the set of higher priority threads relative to the considered subchain $\gamma_{x,h}$ as $\mathcal{T}_k^{\text{hep}}(\gamma_{x,h}) = \{\tau_j \in \mathcal{T}_k | \pi_j \geq \pi_{x,h}\}$, where $\pi_{x,h} = \min\{\pi_i \mid \tau_i \in \gamma_{x,h}\}$. Therefore, the set $\mathcal{T}_k^{\text{hep}}(\gamma_{x,h})$ includes all the threads in P_k with priority higher than or equal to at least one of the threads in the chain.

A thread instance is said to be *carried in* at time t if it is pending both at time t and at time $t-1$. Similarly, a time instant t is a *quiet time* for a partition P_k and a subchain $\gamma_{x,h}$ if no thread instances of threads $\tau_h \in \mathcal{T}_k^{\text{hep}}(\gamma_x)$ allocated to P_k are carried in. An interval $[t_1, t_2)$ is a *busy window* for a subchain instance of $\gamma_{x,h}$ if both t_1 and t_2 are quiet times for P_k, no quiet time occurs in (t_1, t_2), and the subchain instance is released in $[t_1, t_2)$. A subchain instance is released when an instance of its first thread is released.

We further define the request-bound function of a thread τ_i as $rbf_i(\Delta) \triangleq \eta_i(\Delta) \cdot e_i$. Given a set of threads $\mathcal{T}'$, its *cumulative request-bound function* is defined as $RBF(\mathcal{T}', \Delta) \triangleq \sum_{\tau_h \in \mathcal{T}'} rbf_h(\Delta)$.

Lemma 1. *Assume $A \geq 0$ is the time, relative to the beginning of an arbitrary busy window under analysis, in which an instance of a subchain $\gamma_{x,h} = \{\tau_s, \ldots, \tau_e\}$ is released, i.e., an instance of its first thread τ_s is released. If $R_{x,h}(A)$ be the least positive value satisfying*

$$sbf_k(A + R_{x,h}(A)) \geq rbf_e(A+1) + RBF(\mathcal{T}_k^{hep}(\gamma_{x,h}) \setminus \{\tau_e\}, A + R_{x,h}(A) + 1), \tag{1}$$

then $R_{x,h} = \max\{R_{x,h}(A) \mid A \geq 0\}$ is a response-time bound for $\gamma_{x,h}$.

Proof. Consider an arbitrary subchain instance under analysis, released at time A. It can be delayed by: **(i)** previously released instances of the same subchain, **(ii)** other subchains running in the same partition, **(iii)** due to lack of supply due to APS. The time interval in which interfering instances can be released are graphically shown in Figure 6. Those due to (i) can be decomposed in two mutually-exclusive contributions, due to: **(a)** the last thread τ_e in the subchain, and **(b)** to other threads in the chain, respectively.

Interference due to the last thread in the chain is bounded by $rbf_e(A+1)$ since instances of the same thread are processed in order of arrival. Therefore, subsequent instances of the last thread τ_e become ready only when the one under analysis completes, thus excluding self-interference due to instances released after A in the arbitrary busy window under analysis.

Interference due to (ii) and (b) can instead be due to instances released in the whole interval $[0, A+R_{x,h}(A)]$. Each thread with higher or equal priority than the lowest priority of a thread in $\gamma_{x,h}$ (and different from τ_e considered before) can delay at least one of the threads in $\gamma_{x,h}$.

These threads are contained in the set $\in \mathcal{T}_k^{\text{hep}}(\gamma_{x,h}) \setminus \{\tau_e\}$, which also includes the self-interference due to threads of the same chain $\gamma_{x,h} \setminus \{\tau_e\}$ (again, and different from τ_e considered before).

Since each interfering thread $\tau_h \in \gamma_{x,h} \setminus \{\tau_e\}$ interferes up to $rbf_h(A+R_{x,h}(A)+1)$ time units, the interference due (ii) and (b) is bounded by $RBF(\mathcal{T}_k^{\text{hep}}(\gamma_{x,h}) \setminus \{\tau_e\}, A+R_{x,h}(A)+$

1). Delays due to (iii) are considered in the supply-bound function, which lower-bounds the minimum service provided in $[0, A + R_{x,h}(A))$. The lemma follows. $\square$

In essence, Lemma 1 converges when the guaranteed minimum service matches the maximum total demand of the subchain under analysis and the interfering threads. Lemma 1 bounds the response time of a subchain instance released at a given point in time A, and it requires to check the condition in an impractical continuum (i.e., all times $A \geq 0$). Therefore, practical applying the condition requires both a bound and a discretization on the analysis interval.

To this end, we assume the busy window to be bounded; if that is not the case, e.g., when the processor is overloaded, no response-time analysis is possible. This assumption is analogous to assuming an overall utilization that does not exceed 100% of the partition capacity [18].

Similar to [17]–[19], the search interval can be bounded with the smallest positive value satisfying the following inequality:

$$sbf_k(A^*) \geq RBF(\mathcal{T}_k^{\text{hep}}(\gamma_{x,h}), A^*), \tag{2}$$

where $RBF(\mathcal{T}_k^{\text{hep}}(\gamma_{x,h}), A^*)$ upper bounds the total workload released in $[0, A^*]$, and $sbf_k(A^*)$ lower bounds the service provided by the partition $\mathcal{P}_k$ in the same interval. Intuitively, Equation (2) follows because, if busy windows are bounded, no thread (and thus no subchain) can have a relative release time longer than A^*, since there is no busy window longer than A^* [18].

By observing that, as in [17], in Equation (1) only term $rbf_e(A + 1)$ depends only on A (while $RBF(\mathcal{T}_k^{\text{hep}}(\gamma_{x,h}) \setminus \{\tau_e\}, A + R_{x,h}(A) + 1)$ and $sbf_k(A + R_{x,h}(A))$ depends on the whole interval), we can define a discrete search space for activation offsets of $\gamma_{x,h} = \{\tau_s, \ldots, \tau_e\}$ analogously as in prior works [17], i.e.,

$$\mathcal{A}_{x,y} \triangleq \{A \mid 0 \leq A \leq A^* \land rbf_e(A + 1) \neq rbf_e(A)\} \cup \{0\}. \tag{3}$$

B. Deriving a Supply-bound Function for APS

Recall that the APS scheduler is composed of a set of partitions $\mathcal{P} = \{P_1 \ldots P_s\}$, where all partitions share a common accounting window with length W. Each thread $\tau_i \in \mathcal{T}$ in the system, belongs to one unique partition. In the following, we seek to find a *lower bound* on the processor time available to partition P_k in time interval Δ. We proceed as follows. First, we discuss the conditions required to ensure that each partition can correctly deliver the promised budget to assigned threads, i.e., ensuring the so-called *global* schedulability of partitions (similar to the context of reservation servers [1]). Second, we target the real-time guarantees internal to each partition. To this end, we define and prove the definition of a supply-bound function for a system without budget reclaiming enabled (Sec. IV-B2). Third, we show that the same definition is safe also when enabling budget reclaiming (Sec. IV-B3).

1) Guaranteeing the budget provisioning: We start posing a fundamental building block for our analysis: the condition required to ensure that none of the cores are overloaded as a result of a wrong budget configuration. This is required by the local schedulability analysis, which needs that the promised budget supply is actually delivered to the partition, which may not happen if the overall partitions' demand is higher than the time available to the core. Lemma 2 shows that this can be ensured as long as the sum of the budgets of partitions allocated to a core is less than the window size [10].

Lemma 2. *Let $\mathcal{P}_j$ denote the set of partitions allocated to core c_j. Then, a core c_j is not overloaded if:*

$$\sum_{P_k \in \mathcal{P}_j} B_k \leq W. \tag{4}$$

In this case, each partition $P_k \in \mathcal{P}_j$ can correctly deliver B_k units of supply to pending workloads every W time units.

Proof: By contradiction, assume that Equation (4) is satisfied, but there exists a partition $P_k \in \mathcal{P}_j$ that delivers less than B_k units of supply every W time units to pending workloads. In this case, it means that either: **(i)** the budget did not increase as the accounting window advanced, or **(ii)** there are other partitions in $\mathcal{P}_j$ that exceeded its assigned budget in the accounting windows, or **(iii)** APS was not able to deliver B_k units of supply to each $P_k \in \mathcal{P}_j$ with pending workloads in the accounting window. Case (i): is impossible due to rule **R3**. Case (ii): is impossible because: **(a)** by rules **R2**, **R8**, and **R4** and **R5**, the budget is correctly decremented (**R2**) and the running partition is immediately descheduled when it depletes its current budget (**R8**) if there is other pending workload (**R4** and **R5**). Case (iii): APS is not able to deliver the assigned budget if the sum of the budgets B_k due to partitions $P_k \in \mathcal{P}_j$ allocated on c_j is greater than the accounting window size. Since each accounting window has length W, this is impossible because Equation (4) is satisfied by assumption. The lemma follows. $\blacksquare$

2) Analysis without considering budget reclaiming: We start defining $sbf_k(\Delta)$, which is graphically shown in Figure 7.

Definition 1. *The supply-bound function of an arbitrary partition $P_k \in \mathcal{P}$ is defined as:*

$$sbf_k(\Delta) \triangleq \left\lfloor \frac{\Delta}{W} \right\rfloor B_k + \max(0, (\Delta \bmod W) - (W - B_k)).$$

Next, we leverage two simple properties of an APS partition.

Property 1. *[APS Service] An APS partition provides at most ϵ time units of service in an interval of length ϵ, i.e., $\forall \Delta \geq 0, \epsilon \geq 0,\ sbf_k(\Delta + \epsilon) \leq sbf_k(\Delta) + \epsilon$.*

Furthermore, we note that a supply-bound function for an APS partition is *super-additive* [20].

Property 2. *[Super-additivity] The supply-bound function is super-additive, i.e., $\forall \Delta \geq 0, \epsilon \geq 0,\ sbf_k(\Delta + \epsilon) \geq sbf_k(\Delta) + sbf_k(\epsilon)$.*

Finally, Property 3 establishes the minimum supply provided in an interval of time with length equal to the window size W.

Property 3. *[Service in a budgeting window] If $\Delta = W$ and the system is not overloaded, it holds $sbf_k(W) = B_k$.*

Proof: The property follows trivially as a corollary of Lemma 2. ∎

Leveraging Properties 1, 2 and 3, Lemma 3 shows that $sbf_k(\Delta)$ as defined in Definition 1 indeed lower bounds the service time provided by an APS partition for the case in which budget reclaiming is disabled.

Lemma 3. *Let P_k be an arbitrary APS partition. If budget reclaiming is disabled then the minimum amount of service provided by a APS partition is lower-bounded by $sbf_k(\Delta)$ as defined in Definition 1.*

Proof: We first show that for any $\Delta \leq W$, $sbf_k(\Delta) = \max(0, B_k - W + \Delta)$. Recall from Property 1 that an APS partition provides no more than one time unit of supply per time unit. Therefore, it holds that $\forall d \geq 0, n \geq 0, sbf_k(d+n) \leq sbf_k(d) + n$. With $W \geq d$ and $n = W - d$ this implies $sbf_k(W) \leq sbf_k(d) + W - d$, which is equivalent to

$$sbf_k(d) \geq B_k - W + d$$

since $sbf_k(W) = B_k$ (Property 3). The supply is also trivially lower-bounded by zero. Therefore, $sbf_k(\Delta) = \max(0, B_k - W + \Delta)$ lower-bounds supply for any $\Delta \leq W$.

For $\Delta > W$ we exploit super-additivity (Property 2). Since $\lfloor \frac{\Delta}{W} \rfloor W + (\Delta \bmod W) = \Delta$, it holds that

$$\begin{aligned} sbf_k(\Delta) &\geq \left\lfloor \frac{\Delta}{W} \right\rfloor sbf_k(W) + sbf_k(\Delta \bmod W) \\ &= \left\lfloor \frac{\Delta}{W} \right\rfloor B_k + \max(0, B_k - W + (\Delta \bmod W)) \end{aligned}$$

∎

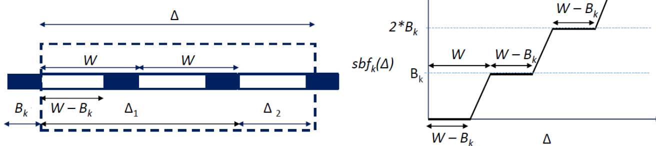


Fig. 7: Two representations of the supply-bound function. The left side denotes the budget usage pattern to minimize the supply in the interval Δ. Here $\Delta_1 = \lfloor (\Delta/W) \rfloor \cdot W$ and $\Delta_2 = \Delta \mod W$ is the rest of the window. The shaded region shows a possible budget usage to reduce the supply in the window under consideration. The right side shows how the supply increases over a period of time.

Thus Lemma 3 captures the supply available to partition P_k in an interval of length Δ. The first portion of the equation refers to the integral units of budget received in each of the budgeting windows, while the second portion captures the partial budget received. It can be noticed that the partial budget computation assumes that we receive the budget at the far end of the interval to compute the minimum supply. If the fractional portion of the interval $(\Delta \mod W) < W - B_k$, then the partition receives no budget in that fractional part.

An alternative representation is shown in Figure 7 where $\Delta_1 = \lfloor (\Delta/W) \rfloor \cdot W$ refers to the region where integral units of budget are received and Δ_2 refers to the region where a fractional budget is received.

As an example, let us consider partition P_k with $B_k = 3$, $\Delta = 28$, and a window size of $W = 10$. Then the partition receives two full budget allocations of worth 6 in the first 20 time units. Later in the tail end of 8 time units, it receives no budget in the first $(10 - 3) = 7$ time units and another unit later. In other words the partition receives $3 - (10 - 8)$ time units in the tail end of the interval. Finally the total budget is 7 units.

It must be pointed that in the supply bound formulation, the *silent period* of $(W - B_k)$ can be a source of pessimism, but given the nature of compositional analysis [2], which relies on only information regarding the current partition, such an overhead is difficult to avoid, and is normally amortized over longer window periods.

3) Analysis considering budget reclamation: We now consider the case in which each partition P_k can consume, in addition to its normal budget B_k, a maximum of $I_k < (W - B_k)$ idle time units, when there are no ready threads in other partitions. This case occurs when the system is under-loaded and therefore partition P_k consumes an extra idle time budget of I_k in its previous window and must pay it back in the interval under analysis.

To start, Lemma 4 proves that the maximum length of the silent period is still bounded by $W - B_k$ as in Lemma 3.

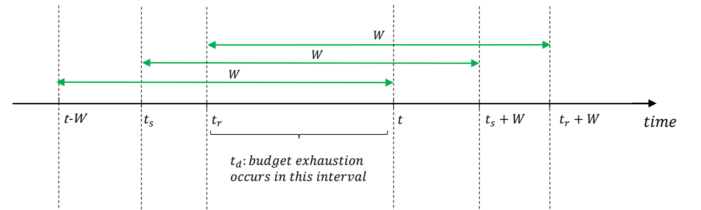


Fig. 8: Relevant times for the proof of Lemma 4

Lemma 4. *Let P_k be an arbitrary APS partition with budget B_k, and $(t, t + \Delta]$ be the window under analysis. Then if P_k consumes an additional idle-time budget $0 < I_k \leq (W - B_k)$ in the last accounting window $(t - W, t]$, the maximum length of the silent period experienced at the beginning of $(t, t + \Delta]$ is bounded by $W - B_k$.*

Proof: Consider an arbitrary interval $(t, t+\Delta]$. There are two mutually-exclusive cases: **(i)** $b_k(t) > 0$ and **(ii)** $b_k(t) \leq 0$. In case (i), the silent period has length 0. Consider now case (ii). Let $t_s \in (t - W, t]$ (shown in Figure 8) be the first time in which a thread of P_k executes in the previous accounting window. Also let $t_r \in (t_s, t - B_k)$ be the time instant in

which a thread of P_k starts executing the last B_k units of time in the last accounting window. That means that the idle time consumed, in effect is $I_k = t_r - t_s$. Since, by assumption, P_k used additional idle-time budget in $(t - W, t]$ it is not descheduled when its current budget reached 0 (rule **R6**), and therefore its current budget became smaller than zero (rule **R2**). The first budget replenishment starts then at $t_s + W$ (rule **R3**), but the current budget is negative (and equal to $-(t_r - t_s)$) and remains smaller than or equal to zero up to when all the idle time has been paid back, i.e., up to time $t_r + W$. At time $t_r + W$, $b_k(t_r + W) > 0$.

The length of the silent period of P_k is given by the difference from time $t_r + W$ and the time $t_d \in (t_r, t]$ in which P_k depletes the budget, i.e., $t_r + W - t_d$. To maximize the length of the silent period experienced at the beginning of $(t, t+\Delta]$ we need to maximize the amount of time the intervals $(t_d, t_r+W]$ and $(t, t+\Delta]$ overlaps. This happens when $t_d = t$. Then we need to chose t_r so that to maximize $t_r + W - t_d$. By construction $t_r \in (t_s, t - B_k]$, and hence we set $t_r = t - B_k$, and we obtain $t_r + W - t_d = t - B_k + W - t = W - B_k$, proving the lemma.

■

With Lemma 4 in place, Property 1 still holds when budget reclaiming is enabled, and therefore the same arguments of Lemma 3 can be applied to show that Definition 1 still holds even when considering budget reclamation.

C. Handling quantization errors in budget accounting

As discussed in Section III-C, budget accounting is performed only in correspondence of the periodic tick ISR and when a system call triggering a scheduling event is called. As a result, budget overruns with a time granularity up to the period T_{ISR} (set to 1 ms by default) of the tick ISR are possible. To keep the analysis simple, we excluded this possibility by means of rule **R8**, which assumes partitions to be immediately descheduled in correspondence of budget exhaustion. Furthermore, mitigation strategies based on reducing T_{ISR} are discussed in Section III-C. Other pragmatic countermeasures are possible, e.g., based on accounting for a "safety margin" when ensuring Equation 4, (e.g., $\sum_{P_k \in \mathcal{P}_c} B_k \leq \alpha \cdot W$, with $\alpha = 0.95$) to account for this and other overheads and variances to the theoretical model (e.g., in the WCETs' accuracy that are often hard to estimate in multicore platforms). Similar precautions are taken in other schedulers, e.g, `SCHED_DEADLINE` in Linux. Alternatively, the derivation of the supply-bound function can be extended to pessimistically account that every time a high-priority thread running in a partition starts executing, it can execute T_{ISR} units of time more than expected due to budget quantization. However, both this and the derivation of a fine-grained bound that is aware of the budget quantization phenomenon is left as future work.

V. Evaluation

This section presents the results of two evaluation studies we performed to evaluate the proposed analysis. We start the discussion by considering a relatively simple synthetic application; then, we report the results of an evaluation we performed on an autonomous construction vehicle case study. To this end, we implemented a prototype of our analysis on top of the pyCPA framework [21]. Furthermore, we also run the same event chains considered by the analysis on a real platform consisting of a Raspberry Pi 4B with 4 CPUs and 4GB RAM and the QNX 7.1 Software development Platform.

A. Evaluation on a Synthetic Application

To evaluate our approach, we first study a relatively simple example case before we apply it to a larger industrial case study. In the first set of experiments we study a simple system consisting of three threads, τ_1, τ_2, and τ_3 (see Table VII). All threads are allocated to the same core. Two chains are specified, $\gamma_1 = (\tau_1, \tau_2)$ and $\gamma_2 = (\tau_3)$. γ_1 is allocated to a partition P_1 and γ_2 to a partition P_2.

TABLE VII: Example Application

	WCET [ms]	**Period [ms]**	**Priority**	**Core**	**Partition**
τ_1	20	100	255	1	P_1
τ_2	10	–	254	1	P_1
τ_3	40	100	253	1	P_2

Analysis-driven design of the budgets. Different budget assignments are evaluated first by running the analysis. For this, the budget of P_1 is gradually increased while the remaining budget is assigned to P_2. As visible in Fig. 9, solutions are found for budget values of P_1 in the range from 32% to 58%. As expected, an increasing budget for P_1 has a significant effect on the latency of γ_1 while the latency of γ_2 increases only marginally. This is because γ_2 consist of only a single thread. This highlights the benefits of our analysis, which allows finding non-trivial trade-offs in the design of real-time applications running on QNX, without requiring to actually deploy the system with different budget values and observe the results, and providing analytical guarantees for the application timing constraints that would not be possible to ensure by just running the application.

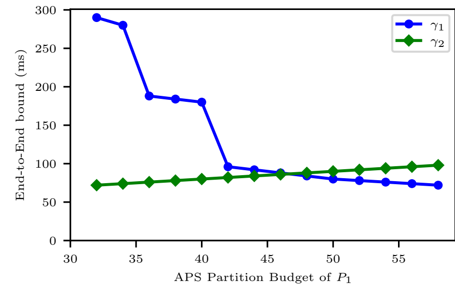


Fig. 9: End-to-End bound of γ_1 and γ_2 with varying budget.

Comparing measurement against analysis results. In this experiment, we compare the analytical bound on the latency of γ_1 against the largest observed values within experiment runs of 20 s on the QNX platform in the standard configuration with budget reclaiming enabled. γ_1 is deployed on a core of the platform within its partition P_1. A second partition is added

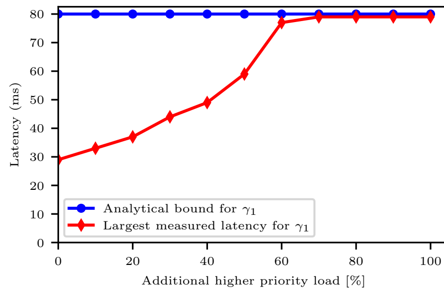


Fig. 10: Comparison of observed latency for γ_1 against the analytical worst-case bound and varying higher priority load.

to the same core that consumes the remaining capacity. This partition includes another interference inducing load thread with a period of $10\,\text{ms}$ that is used to consume all the idle time that would otherwise be available to γ_1. In different experiment runs, the execution time of the higher priority load thread is increased from $1\,\text{ms}$ to $10\,\text{ms}$ and the largest observed latency of γ_1 is recorded.

Fig. 10 presents the results. When the interference by the other thread is low, the measured latency of γ_1 is lower than the one predicted by the analysis. Indeed, thanks to idle time reclamation, γ_1 is allowed to use more budget than the assigned one. As the load thread demands more time, the observed latency values increases, and it converges to the analytical bound once the load increases to 60%, demonstrating the tightness of our approach for this application.

B. Autonomous Construction Vehicle Case Study

The case study investigates a tactical decision-making system of an autonomous construction vehicle [22]. The system implements a behavior generation and trajectory planning algorithm for operation in dynamic environments such as construction sites. It identifies traffic participants, considers their likely future behavior, and selects a suitable response out of a set of possible actions.

The application consists of 11 threads, as described in Table VIII. The source thread is triggered periodically with a period of 300 ms. The original application is designed for a single-core platform and executes all threads sequentially. However, the application can be parallelized to allow deployment on multiple cores of a multi-core platform without altering its functionality, as shown in Fig. 11. The main timing requirement is specified for the chain γ_1 that spans from e_{in} to e_{out} and has an end-to-end deadline of $600\,\text{ms}$.

The application is divided into five sub-chains $\gamma_{1,1}$ to $\gamma_{1,5}$. Each sub-chain is assigned to a dedicated APS partition. The APS scheduler is configured with a window size of $100\,\text{ms}$, and each APS partition is configured with its budget according to Table IX. By running our analysis, the application designer is then able to determine whether this configuration (i.e., priorities, threads-to-core and threads-to-partition allocation, and budgets) allows meeting the timing constraint of $600\,\text{ms}$, by computing a bound on the end-to-end latency of $421.8\,\text{ms}$.

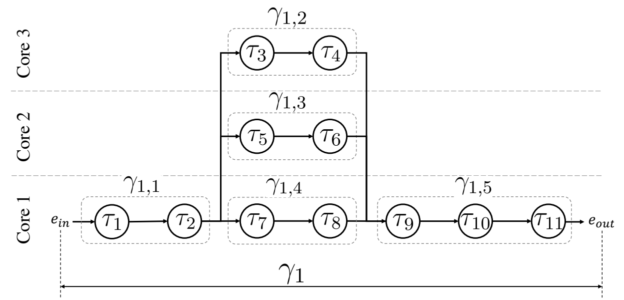


Fig. 11: Use Case: Tactical decision making system of an autonomous construction vehicle.

TABLE VIII: Threads of the Case Study

	Name	**WCET**	**Prio**	**Core**	**Part**
τ_1	MPDM	2,2	250	1	P_1
τ_2	Sim_Crowd	0,1	249	1	P_1
τ_3	Move_Vehicle	12	248	3	P_2
τ_4	Vehicle_Force	169,1	247	3	P_2
τ_5	Move_Pedestrian	3,3	246	2	P_3
τ_6	Pedestrian_Force	85,5	245	2	P_3
τ_7	Follow_Vehicle	0,1	244	1	P_4
τ_8	Follow_Force	6,4	243	1	P_4
τ_9	Stop_Vehicle	0,1	242	1	P_5
τ_{10}	Cost	1,2	241	1	P_5
τ_{11}	Optimal_Policy	0,1	240	1	P_5

Analysis-driven design of the budgets. This experiment evaluates the end-to-end latency of subchain γ_1 when the allocated budget is varied. Each curve in Fig. 12 is recorded by varying the budget allocated to the specific partition from 5% to 100% in steps of 5%, while all other partitions have a budget of 100% and the APS window size is set to $100\,\text{ms}$.

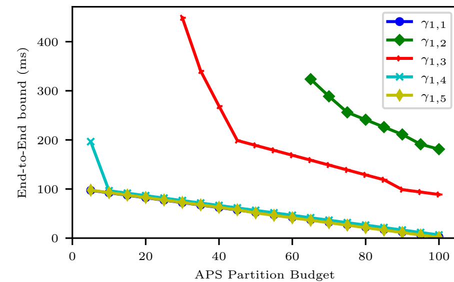


Fig. 12: Impact of varying budget for each subchain on the end-to-end latency, for the case-study.

The results are reported in Fig. 12. A large effect on the resulting latency can be seen for $\gamma_{1,2}$ and $\gamma_{1,3}$ as these subchains contain threads with high execution times. $\gamma_{1,1}$ and

TABLE IX: Configured Partitions for the Case Study

	Window [ms]	**Budget Global**	**Budget Core**
P_1	100	64%	16%
P_2	100	80%	20%
P_3	100	44%	11%
P_4	100	12%	3%
P_5	100	14%	3,5%

$\gamma_{1,5}$ show nearly identical results. These subchains contain only very light threads, and the main effect on the latency, therefore, comes from the silent period $W - B_k$

Running the case study on a real platform. In the last experiment, the case study is deployed on the real platform, configuring the APS scheduler with and without budget reclaiming, and the end-to-end latency is recorded for consecutive instances of γ_1 in a timespan of 20 s.

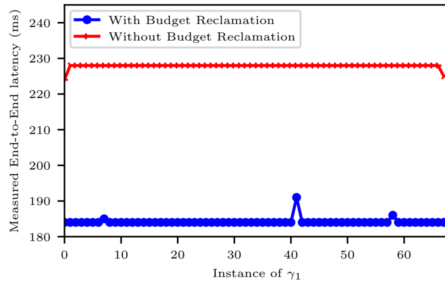


Fig. 13: Observed latency of consecutive instances of γ_1 on the hardware platform.

Fig. 13 presents the measurement results. If budget reclamation is not enabled, the observed latency is 228 ms. If budget reclamation is enabled, chain γ_1 can use the idle time to execute, completing earlier and allowing to reduce the end-to-end latency to 184 ms. There is a small spike at $t = 40$, where the latency goes up. This is attributed to the overhead imposed by logging functionalities and other OS-related activities that cause additional contention on the cores.

VI. Related Work

To the best of our knowledge, no previous work provided a response-time analysis and design principles for real-time applications running on QNX. The only work closely related work is due to Dasari et al. [10], who experimentally compared different configurations of QNX in simulation, and showed that some of them may lead to unpredictable response times.

In the last years, a lot of attention has been given to the problem of studying the end-to-end response time of event chains, which are also studied in this work. Most closely to us, Casini et al. [17] provided a response-time analysis for chains running on ROS 2 using resource reservations. Blass et al. [7] proposed an automated method for using the ROS 2 analysis to select the parameters required by the resource reservation mechanism implemented by the `SCHED_DEADLINE` scheduling class [6] of Linux. Both of them used supply-bound functions to model the minimum amount of service provided by the system: we believe that the results of this paper may also be used as a building block for deriving an analysis for ROS 2 applications running on QNX. Other works proposed analyses or scheduling mechanisms for ROS 2 [19, 23, 24] and other middlewares affecting scheduling [25]–[28].

Concerning the analysis of event chains, multiple research directions have been followed in the past. This paper builds upon the CPA [9, 29] techniques, for which multiple extensions have been proposed over the years [30, 31]. Thiele et al. [32] proposed the real-time calculus, an analysis approach based on network calculus [20] sharing some aspects of CPA. Tindel et al. [33, 34] proposed a holistic response time analysis that considers transactions (sequential task chains) that can spread over multiple processing nodes, connected by a network. The methods has been later extended to account for dynamic offsets [35], precedence constraints [36], to improve the analysis accuracy [37] and to support hierarchical time partitioned scheduling [38]. Becker et al. [39] proposed methods to bound the maximum data age of cause-effect chains in automotive systems, also considering the Logical Execution Time (LET) paradigm [40, 41]. A method to compute the different end-to-end delays of multi-rate cause-effect chains is presented in [42], however, the authors do not focus on reservations. Another research direction studied the timing behavior of time-triggered [43]–[46] (i.e., asynchronous) chains either considering implicit communication or using the LET paradigm. Precedence constraints have also been studied in the context of parallel real-time tasks [47]–[50]. Most relevant to us are those works targeting partitioned scheduling [51]–[53], which, however, do not support DAGs with multiple sources triggered at different rates and a per-thread priority assignment. Finally, a relevant branch of related research consists in the study of resource reservation mechanisms: many different reservation algorithms have been proposed over the years (e.g., [54, 55]) working under different scheduling schemes. One of the seminal works is due to Shin and Lee [2], who proposed the periodic resource model, which has been used as a basis for many subsequent derivations of supply-bound functions [56]–[58].

VII. Conclusions

This paper proposed a deep investigation of the real-time behavior of the APS scheduler of QNX, proposing a model for event chains running in QNX under APS. The model has been validated with an extensive set of experiments. By leveraging the model, a real-time analysis to bound the end-to-end delay of event chains has been proposed. Furthermore, we presented the results of an experimental evaluation we performed to evaluate different design strategies on a real autonomous construction vehicle case study. In addition, we discussed some practical issues related to the time accounting in QNX, and on the way precedence constraints can be correctly implemented when working with APS.

Interesting research directions for future work include the investigation of priority assignment schemes [59]–[61], the consideration of time-driven chains [45], e.g., under the LET paradigm [46], the comparison with other schedulers like `SCHED_DEADLINE`, and the derivation of optimization strategies to simultaneously optimize budgets, priorities, and the threads-to-core allocation.

REFERENCES

[1] A. Biondi, G. C. Buttazzo, and M. Bertogna, "Schedulability analysis of hierarchical real-time systems under shared resources," *IEEE Transactions on Computers*, vol. 65, no. 5, pp. 1593–1605, 2016.

[2] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *24th IEEE Real-Time Systems Symposium*, 2003.

[3] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.

[4] F. Farshchi, Q. Huang, and H. Yun, "Bru: Bandwidth regulation unit for real-time multicore processors," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[5] Blackberry QNX, *Adaptive Partitioned Scheduler User Guide – QNX® Software Development Platform 7.1*, 2021.

[6] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the Linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016.

[7] T. Blass, A. Hamann, R. Lange, D. Ziegenbein, and B. B. Brandenburg, "Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems," in *Proceedings of the 27th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[8] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Third Edition*. Springer, 2011.

[9] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the SymTA/S approach," *IEEE Proceedings - Computers and Digital Techniques*, March 2005.

[10] D. Dasari, A. Hamann, H. Broede, M. Pressler, and D. Ziegenbein, "Brief industry paper: Dissecting the qnx adaptive partitioning scheduler," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021, pp. 477–480.

[11] A. Biondi and Y. Sun, "On the ineffectiveness of 1/m-based interference bounds in the analysis of global edf and fifo scheduling," *Real-Time Systems*, vol. 54, no. 3, pp. 515–536, 2018.

[12] B. B. Brandenburg and M. Gül, "Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 99–110.

[13] "https://www.qnx.com/developers/docs/6.3.0sp3/neutrino/sys_arch."

[14] Blackberry QNX, *QNX® Neutrino® RTOS Programmer's Guide – QNX® Software Development Platform 7.1*, 2020.

[15] "https://lore.kernel.org/lkml/20210208073554.14629-3-juri.lelli@redhat.com/."

[16] Blackberry QNX, *Getting Started with QNX® Neutrino®: A Guide for Realtime Programmers – QNX® Software Development Platform 7.1*, 2020.

[17] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-time analysis of ros 2 processing chains under reservation-based scheduling," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[18] S. Bozhko and B. B. Brandenburg, "Abstract Response-Time Analysis: A Formal Foundation for the Busy-Window Principle," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, vol. 165, 2020, pp. 22:1–22:24.

[19] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response time analysis and priority assignment of processing chains on ros2 executors," in *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2020, pp. 231–243.

[20] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Berlin, Heidelberg: Springer-Verlag, 2001.

[21] J. Diemer, P. Axer, and R. Ernst, "Compositional performance analysis in python with pycpa," in *Proceedings of the Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2012.

[22] M. Gallardo and S. Chakraborty, "Decision making for autonomous construction vehicles," Master's thesis, Mälardalen University, 2019.

[23] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 41–53.

[24] H. Choi, Y. Xiang, and H. Kim, "Picas: New design of priority-driven chain-aware scheduling for ros2," 2021.

[25] D. Casini, A. Biondi, and G. Buttazzo, "Analyzing parallel real-time tasks implemented with thread pools," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19, 2019.

[26] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quiñones, "Timing characterization of openmp4 tasking model," in *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

[27] D. Casini, A. Biondi, and G. Buttazzo, "Timing isolation and improved scheduling of deep neural networks for real-time systems," *Software: Practice and Experience*, vol. 50, no. 9, pp. 1760–1777, 2020.

[28] J. Sun, N. Guan, Y. Wang, Q. He, and W. Yi, "Real-time scheduling and analysis of openmp task systems with tied tasks," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Dec 2017.

[29] J. Rox and R. Ernst, "Compositional performance analysis with improved analysis techniques for obtaining viable end-to-end latencies in distributed embedded systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 3, 2013.

[30] S. Schliecker and R. Ernst, "A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems," in *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '09, 2009.

[31] J. Schlatow and R. Ernst, "Response-time analysis for task chains with complex precedence and blocking relations," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, Sep. 2017.

[32] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings*, May 2000.

[33] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocessing and microprogramming*, vol. 40, no. 2-3, pp. 117–134, 1994.

[34] K. Tindell, "Adding time-offsets to schedulability analysis. department of computer science, university of york," Technical Report YCS-221, Tech. Rep., 1994.

[35] J. C. Palencia and M. G. Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *Proceedings 19th IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 1998, pp. 26–37.

[36] ——, "Exploiting precedence relations in the schedulability analysis of distributed real-time systems," in *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*. IEEE, 1999, pp. 328–339.

[37] J. Mäki-Turja and M. Nolin, "Efficient implementation of tight response-times for tasks with offsets," *Real-Time Systems*, vol. 40, no. 1, pp. 77–116, 2008.

[38] A. Amurrio, E. Azketa, J. J. Gutierrez, M. Aldea, and M. G. Harbour, "Response-time analysis of multipath flows in hierarchically-scheduled time-partitioned distributed real-time systems," *IEEE Access*, vol. 8, pp. 196 700–196 711, 2020.

[39] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *Journal of Systems Architecture*, vol. 80, pp. 104 – 113, 2017.

[40] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *International Workshop on Embedded Software*. Springer, 2001, pp. 166–184.

[41] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the let paradigm," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[42] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A Compositional Framework for End-to-End Path Delay Calculation of Automotive Systems under Different Path Semantics." in *Proceedings of the IEEE Real-Time System Symposium, Workshop on Compositional Theory and Technology for Real-Time Embedded Systems,*, (2008).

[43] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli, "Period optimization for hard real-time distributed automotive systems," in *2007 44th ACM/IEEE Design Automation Conference*, June 2007.

[44] T. Kloda, A. Bertout, and Y. Sorel, "Latency analysis for data chains of real-time periodic tasks," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2018, pp. 360–367.

[45] M. Günzel, K.-H. Chen, N. Ueter, G. von der Brüggen, M. Dürr, and J.-J. Chen, "Timing analysis of asynchronized distributed cause-effect

chains," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 40–52.

[46] P. Pazzaglia, A. Biondi, and M. Di Natale, "Optimizing the functional deployment on multicore platforms with logical execution time," in *2019 IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 207–219.

[47] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011.

[48] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *2010 31st IEEE Real-Time Systems Symposium*, Nov 2010.

[49] J. Fonseca, G. Nelissen, and V. Nélis, "Improved response time analysis of sporadic DAG tasks for global FP scheduling," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS '17, 2017.

[50] Q. He, x. jiang, N. Guan, and Z. Guo, "Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2283–2295, 2019.

[51] F. Aromolo, A. Biondi, G. Nelissen, and G. Buttazzo, "Event-driven Delay-induced Tasks: Model, Analysis, and Applications," in *2021 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2021.

[52] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho, "Response time analysis of sporadic dag tasks under partitioned scheduling," in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, 2016, pp. 1–10.

[53] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "Partitioned fixed-priority scheduling of parallel tasks without preemptions," in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 421–433.

[54] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, Madrid, Spain, December 2-4 1998.

[55] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo, "Iris: a new reclaiming algorithm for server-based real-time systems," in *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium, 2004.*, May 2004.

[56] D. Casini, L. Abeni, A. Biondi, T. Cucinotta, and G. Buttazzo, "Constant bandwidth servers with constrained deadlines," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, ser. RTNS '17, 2017.

[57] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, July 2003, pp. 151–158.

[58] E. Bini, M. Bertogna, and S. Baruah, "Virtual multiprocessor platforms: Specification and use," in *2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 437–446.

[59] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns, "A review of priority assignment in real-time systems," *Journal of systems architecture*, vol. 65, pp. 64–82, 2016.

[60] J. G. García and M. G. Harbour, "Optimized priority assignment for tasks and messages in distributed hard real-time systems," in *Proceedings of Third Workshop on Parallel and Distributed Real-Time Systems*. IEEE, 1995, pp. 124–132.

[61] M. Richard, P. Richard, F. Cottet, D. Dietrich, P. Neumann, and J. Thomesse, "Task and message priority assignment in automotive systems," in *4th FeT IFAC conference on fieldbus systems and their applications*, vol. 15, 2001, p. 16.