On the QNX IPC: Assessing Predictability for Local and Distributed Real-Time Systems

Matthias Becker[1], Dakshina Dasari[2], and Daniel Casini[3]
[1]*KTH Royal Institute of Technology, Stockholm, Sweden*
[2]*Robert Bosch GmbH, Germany*
[3]*TeCIP Institute and Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy*

***Abstract*—With the advent of massively distributed applications such as those required by the IoT-to-Edge-to-Cloud compute continuum (i.e., automotive, smart agriculture, smart manufacturing, and more), real-time communication mechanisms allowing physically distributed nodes to seamlessly communicate as if they were running on the same host acquired noteworthy importance. To this end, the synchronous inter-process communication (IPC) mechanism provided by the QNX operating system (OS) is a promising candidate, as it allows using the application programming interface for communicating both on a single- and multi-node setting. Furthermore, it provides *priority* and *partition inheritance* mechanisms to improve predictability when working with the Adaptive Partitioning Scheduler (APS), a reservation-based scheduler provided by the QNX OS. This paper explores the behavior of the QNX synchronous message-passing (SyncMP) IPC with an extensive set of experiments, using them to formalize its behavior and model it from a real-time perspective. Then, it provides a response-time analysis for client-server applications based on the QNX SyncMP building upon self-suspending task theory. Finally, we evaluate the analysis on an application based on the WATERS 2019 Challenge by Bosch.**

I. Introduction

Modern applications across several domains are predominantly distributed and consist of a set of communicating components located on different nodes, possibly spanning the IoT, to the edge, up to the cloud. A case in point is autonomous driving [1]–[4], that requires intensive computations, which may be offloaded to surrounding edge nodes or the cloud, and could also interact with the surrounding IoT infrastructure. In essence, functionalities are often realized by elements that are executed on the local node, e.g., an electronic control unit (ECU) of a car, and others that are offloaded elsewhere, either another ECU or to the edge/cloud in a client-server fashion. This gives rise to the need for a *compute continuum*, where nodes can seamlessly interact in a distributed environment as if executing on the same node.

To this end, the QNX operating system provides a set of *synchronous message-passing primitives* (SyncMP) for *inter-process communication* (IPC) devised to be used to implement such a client-server paradigm [5]. Unlike most of the other IPC mechanisms, the QNX IPC allows for the seamless integration of components distributed onto multiple nodes by transparently using the same application programming interface (API), via the *QNX Transparent Distributed Processing* (QNET) component. QNET enables logically merging a set of distributed QNX devices into a single logical computer where all applications can communicate through a unified programming interface. Furthermore, QNX is ISO-26262 certified at the highest level of assurance (ASIL-D), making it widely preferred by many automotive OEMs [6].

Another key aspect for most of these applications is the need to satisfy real-time requirements. From this side, QNX is a promising candidate. Indeed, the QNX OS provides the *Adaptive Partitioning Scheduler* (APS) [7], which implements fixed-priority scheduling and a reservation-based mechanism where threads are grouped into "partitions", i.e., virtual containers with a guaranteed fraction of processing bandwidth. The assigned bandwidth is ensured to be delivered irrespective of what is executed in other partitions, and it allows for providing temporal isolation among partitions. This is very desirable in the target distributed setup, where modules running in different partitions can have different levels of criticality.

Furthermore, QNX provides *priority and partition inheritance* mechanisms to improve the predictability of its SyncMP when used with APS. So when a server thread is executing a service on behalf of a client, it can inherit the priority and the partition bandwidth associated with the client. This prevents the priority-inversion phenomenon that would arise if the server executed with a lower priority than the requesting thread and allows for designing servers' partitions more easily.

However, QNX is closed-source, and how APS and the SyncMP mechanism interact is not extensively documented. Furthermore, only a few works studied QNX from a real-time perspective [6, 8] and to our knowledge, none of them explore the local IPC mechanisms or the aforementioned transparent distributed processing from a timing perspective. This makes it hard to provide a suitable model of the system to allow deriving timing bounds for distributed applications using QNX.

Contributions. To fill this gap, this work explores the behavior of the QNX SyncMP mechanism when working in conjunction with APS. To this end, we perform extensive experiments to characterize its behavior from a real-time perspective, and derive a suitable model for analyzing QNX-based client-server applications. Furthermore, we describe the QNX SyncMP and its interactions with APS with a set of rules. The model and the rules are then used for deriving a response-time analysis based on self-suspending task theory [9], which sets the foundation for deriving design methodologies and analysis-driven orchestration mechanisms for distributed applications running QNX in future work. Finally, we report the results of an extensive evaluation we performed to: **(a)** evaluate the communication latency when using SyncMP; **(b)** compare the proposed WCRT bounds with measurements on the platform; and **(c)** evaluate the analysis on a realistic autonomous driving application based on the WATERS 2019 Challenge by Bosch [10].

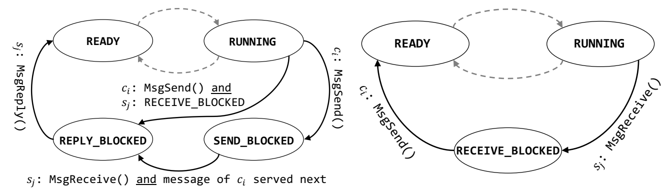
II. QNX OS Overview

This section introduces the main features of the QNX OS that are considered in this work.

The Adaptive Partitioned Scheduler (APS). The APS is a reservation-based scheduler that groups threads into virtual containers called partitions. By budgeting the execution of each partition, QNX APS allows providing each partition with a fraction of the processing capacity. For each partition, its budget determines the amount of processing time in a sliding window, common to all partitions, set to 100 ms by default. When the budget reaches 0, the partition is throttled, and its budget is gradually restored when enough time has passed. The QNX scheduler combines APS with a fixed-priority scheduler, meaning that, at any point in time, the highest-priority thread with a positive budget is selected to run. There are 255 priority levels, with 255 being the highest priority. Partitions are created by subtracting budget from the *system partition*, a parent partition that initially has 100% of the budget. APS also provides the so-called idle-time mode to improve the performance in the average case through budget reclamation. Threads can be assigned to specific cores using *core affinities*.

Inter-process communication in QNX. QNX offers different IPC mechanisms like synchronous message passing (SyncMP), signals, FIFOs, pipes, message queues, and shared memory communication. Among these, SyncMP and signals are implemented in the microkernel, while the others are offered as external services. In this work, we focus on synchronous message passing that forms the foundational IPC message mechanism provided by the QNX OS. This is based on the three key primitives, `MsgSend()`, `MsgReceive()`, and `MsgReply()`, which form the basis to realize the *client-server* paradigm [5]. Fig. 1 shows the kernel state transitions involved when a client and server interact using the SyncMP primitives mentioned earlier. The client (sender) thread invokes a blocking `MsgSend()` until it receives a `MsgReply()` from the server (receiver) thread. The server, on the other hand, listens to incoming messages with a `MsgReceive()`, processes them, and replies with a corresponding `MsgReply()`. As opposed to `MsgSend()`, `MsgReply()` is non-blocking, implying that a server thread could reply to a client and continue with its normal processing while the underlying networking stack (kernel) could asynchronously transfer the reply data to the client thread.

QNX Channels. With channels, QNX offers an abstraction over the regular message passing to be used for client-server communication with the discussed SyncMP primitives. The channel facilitates many-to-one communication, and therefore, multiple threads could connect to a given channel. Every channel records the threads waiting for messages, un-replied



(a) Client States (b) Server States

Figure 1: Kernel states and transitions the client (a) and server (b) transition during message passing. Other state transitions are indicated in dashed lines.

messages, and received but not yet replied messages for bookkeeping. When multiple messages are waiting on the same channel, they are consumed in priority order by a server calling `MsgReceive()`, according to the priority of the client that sent it.

QNX Transparent Distributed Processing (QNET). QNET is the native network manager of QNX. It provides the system developer with a uniform interface to access resources from either local and remote nodes, giving the appearance of a single logical computer to the developer while abstracting the underlying pool of interconnected QNX-based devices. In this work, we explore the behavior of the QNX SyncMP also in distributed setups leveraging QNET.

Priority Inheritance and Server Boost. QNX allows to enable *inheritance*, in the form of *priority* and *partition inheritance*. When inheritance is enabled, server threads inherit the priority of the client they are serving and execute it on behalf of the client. QNX also provide the "Server boost" feature, aimed to minimize processing delays for blocked high-priority threads. So, when a high-priority thread becomes `SEND_BLOCKED` on a server (meaning its requests are queued at the server and the server is not actively handling them), then in an attempt to speed up the processing, the kernel selects candidate server threads that may receive on the given channel (and service the request), and boosts their priorities. Also, when none of the server threads are `RECEIVE_BLOCKED` on a channel, then the kernel boosts the priority of all server threads that last received on that channel, with the intent that they would finish their current processing and service the incoming requests faster with the boosted priority. While this feature could improve the average-case performance, it could generally harm predictability by raising the priority of multiple threads at the same time.

Partition Inheritance. When a server executes on behalf of a client, it could be desirable to bill the corresponding execution cost to the client partition. QNX handles this with the concept of partition inheritance. Server processes can be hosted in zero or minimal-budget partitions, and whenever a server executes on behalf of a client, the execution is billed to the partition of the client. Interestingly, we observed that the QNX implementation leads to some non-intuitive (undocumented) behavior that differs depending on whether the client and the server run on the same node or in different nodes. We explore this in more detail in Section III.

III. System Model

This paper considers a set of m identical cores $\mathcal{U} = \{p_1, \dots, p_m\}$ running the QNX APS scheduler. Cores can be potentially located on different nodes in a distributed system.

A. Workload Model

The workload is composed of a set $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$ of n real-time threads. Each thread is characterized by a priority π_i. Threads are scheduled according to *partitioned scheduling*, i.e., each thread is assigned to only one core. Threads are divided into two categories: *clients* and *servers*. Thus, the application behavior is realized by clients that offload particular functionalities to servers. Each client c_i is characterized by a worst-case execution time (WCET) e_i. Each client thread is released sporadically with minimum inter-arrival time T_i. Each client thread instance needs to complete within $D_i \le T_i$ time units from its release (constrained deadline). Each client $c_i \in C \subseteq \mathcal{T}$ communicates with zero or more servers threads in $S \subset \mathcal{T}$ by requiring zero or more services.

Each server $s_h \in S$ provides one or more *services* $\mathcal{Q}(s_h) \subseteq \mathcal{Q}$, where $\mathcal{Q}$ is the set of all services. Servers are expected to follow the code structure suggested by the QNX documentation [11], shown in Algorithm 1. A server receives all service requests on the same channel, identified by a service type. A service $\sigma_x \in \mathcal{Q}$ is realized by means of the QNX SyncMP, based on `MsgSend()`-`MsgRcv()`-`MsgReply()`. We assume each service is provided by only one server. This assumption allows making the system more predictable by: **(i)** reducing the server boost effect, discussed in Section II, to a single server, thus removing any uncertainty to which server priority is boosted, and **(ii)** allowing to avoid the LIFO scheduling effects that would arise when a service can be handled by multiple servers [5]. Each service $\sigma_x \in \mathcal{Q}$ is characterized by a *worst-case service time* (WCST) $\omega_{x,i}$, bounding the time required to complete service σ_x (without interference) when called by a client $c_i \in C$. The WCST of a service depends on the client issuing the request because, for example, different clients may use different input data, with potentially large effects on the time required to handle the request.

Client WCETs and server WCSTs already include the OS overheads to receive and transmit data.

Furthermore, an interaction between a client c_i and a server s_j for a service σ_x is characterized by a pair of worst-case communication delays (WCCDs) $\delta^x_{i,j}$ and $\delta^x_{j,i}$, bounding the time required to transmit and receive the data through the physical medium, respectively. Whenever such a time is negligible (e.g., if the client and the server is allocated to the same core or node), $\delta^x_{i,j} = \delta^x_{j,i} = 0$. We consider a discrete-time model in which all time parameters are integer multiples of a basic time unit $\epsilon = 1$ (e.g., a processor cycle).

For each pair of client c_i and service σ_x, $n^{\mathrm{S}}_{i,x} \in \mathcal{N}$ is the number of times each instance of c_i requires σ_x. Given a service σ_x, the function $\mathcal{Q}^{-1}(\sigma_x) = s_h$ returns the server providing that service.

Algorithm 1 Expected structure of a server thread in QNX.

```
1: ch_id ← ChannelCreate()
2: while true do
3:     rcv_id ← MsgReceive(ch_id, msg)
4:     switch msg.type do
5:         case type A
6:             ...                      ▷ Handle a service of type A
7:         case type ...
8:             ...                      ▷ Handle a service of type ...
9:     MsgReply(rcv_id, response)
10: end while
```

We assume service requests cannot be nested: i.e., a server handling a request cannot perform any service request to any other server (thus acting as a client thread). Although supporting nested requests can be a case of practical relevance, due to space constraints, we prefer to leave it to future work as the QNX manual itself [5] (page 80) suggests taking special care when supporting nested service requests because they can lead to issues such as deadlocks. Their study would then require extended considerations for the modeling and analysis. The worst-case response time (WCRT) R_i of a client c_i is the longest time span from the release to the completion of any of its instances. A client c_i is said to be schedulable if $R_i \le D_i$.

B. Adaptive Partitioning Scheduler Model

The system includes a set of APS partitions $\mathcal{P} = \{P_1, \dots, P_s\}$. Each partition P_k is characterized by a nominal budget of B_k time units. The symbol $b_k(t)$ denotes the current budget of P_k at time t. The accounting window size is denoted with W [6]. As in prior work [6], we restrict to the more predictable case [12] in which also partitions follow partitioned scheduling, i.e., each core $p_j \in \mathcal{U}$ can host multiple partitions, referred to as $\mathcal{P}_j$, but each partition is managed by one core only. For any arbitrary thread (either a client or a server) τ_i, $\mathrm{hep}^{\mathrm{cl}}_k(\tau_i)$ and $\mathrm{hep}^{\mathrm{sr}}_k(\tau_i)$ denote the sets of *all clients* and *all servers* with higher or equal priority than τ_i in the same partition P_k, respectively, excluding τ_i. $\mathrm{hep}^{\mathrm{cl}}(\tau_i)$ and $\mathrm{lp}^{\mathrm{cl}}(\tau_i)$ include all clients from all partitions with higher-or-equal and lower priorities, respectively, independent of the partition. Set S_k denotes the set of all the servers in partition P_k. We model the supply-time given by an arbitrary partition P_k using the supply-bound function abstraction [13]–[15], in which $sbf_k(\Delta)$ bounds the minimum amount of supply provided by partition P_k in any interval of length Δ. A supply-bound function instance for APS has been derived in [6]. The supply-bound function of the system partition for threads allocated on core p_k is denoted with $sbf_{sys,p_k}(\Delta)$.

C. Client-Server Interaction

Next, we formalize the behavior of the client-server interaction of QNX by a set of rules. The first seven rules (**R1-R7**) model the baseline behavior of the mechanism without priority and partition inheritance enabled. This scenario is referred to as `C-S` (i.e., client-server). Rule **R8** extends `C-S` to account for *priority inheritance*. Rule **R9-A** extends **R1-R8** to account for *partition* inheritance in the case in which the client and the server are in the same physical node (called `LOCAL-I`, local with inheritance). Rule **R9-B** extends **R1-R8** to account for *partition* inheritance in the case in which the client and the server are distributed in two different nodes (called `DISTR-I`, distributed with inheritance).

Basic IPC-SyncMP Rules (Scenario C-S):

- **R1:** When an instance of each client thread is released, it is in the `READY` state.
- **R2:** When a client requires a service σ_s it calls the `MsgSend()` system call. Then, it suspends moving to the `SEND_BLOCKED` state.
- **R3:** If a client sends a message to a server that is blocked in the `RECEIVE_BLOCKED` state, the server is awakened and transits to the `READY` state.
- **R4:** When a server calls the `MsgReceive()` system call, if there is no message to be managed, it transits to the `RECEIVE_BLOCKED` state.
- **R5:** After receiving a message (using `MsgReceive()`), the server manages the request by executing the corresponding code (see Algorithm 1), and replies to the client using `MsgReply()`. The client then transits to the `READY` state. While the server handles a request of a client, the client transitions to the `REPLY_BLOCKED` state.
- **R6:** Once a server starts processing a request, the request is served to completion.
- **R7:** When multiple messages are waiting on the same channel, they are consumed in priority order by the server, according to the priority of the client that sent each message. Consequently, services are handled in priority order by servers.

Additional rule for the LOCAL-I and DISTR-I Scenario:

- **R8:** When a service request (i.e., a message) is inserted in the queue of a server, if the priority of the corresponding client is higher than the current server priority, then the server inherits such priority. When a server thread s_i finishes serving a service σ_x, it remains with the same priority until either: **(i)** a new, higher-priority service request is enqueued in s_i, thus raising its priority, or **(ii)** the next call to `MsgReceive()` selects a lower-priority service request, and the priority of s_i is downgraded.

Additional rule for the LOCAL-I Scenario:

- **R9-A:** If a server s_i and client c_j are in the same node, when s_i processes a request from c_j, s_i inherits the partition of c_j to serve the request of c_j, i.e., it consumes the budget of the partition of c_j.

Additional rule for the DISTR-I Scenario:

- **R9-B:** If a server s_i and client c_j are in two different nodes, when s_i processes a request from c_j it uses the budget of the system partition.

D. Rule Validation

Most of the content in the rules we presented is based on a detailed inspection of the QNX manuals. However, the QNX OS is closed-source, and manuals do not contain enough details to enable accurate modeling. Therefore, to complement and corroborate our findings with empirical evidence, we designed a specific set of experiments, and we ran them on a real platform.

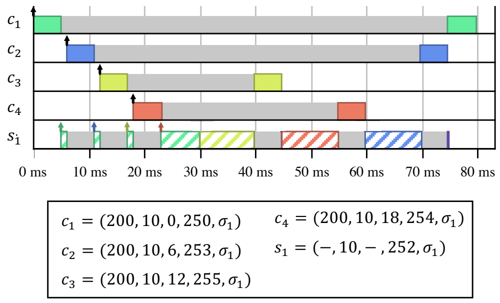
All experiments are performed on a Raspberry Pi 4B with 4 cores and 4GB RAM using the QNX 7.1 Software Development Platform (SDP). For each of the evaluated settings, the QNX event tracing facility is used to obtain the execution trace. The kernel instrumented for tracing operates typically 98% as fast as the non-instrumented kernel [5]. During tracing, events are stored in a buffer inside the kernel, from where they are read by a data capture utility. In our experiments, event filters are set such that only relevant events from our application are recorded. We further extended the tracing facilities to record detailed information on APS partition usage over time. Where applicable, we log the APS partition statistics at a granularity of 1 ms. Furthermore, the data capture utilities are allocated to a core that is not used for the experiments.

QNX APS does not allow enabling or disabling priority and partition inheritance separately: therefore, in the following, we simply refer to enabling or disabling inheritance to mean both.

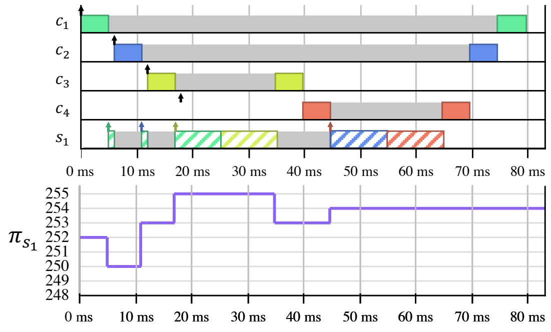
1) Experiments on a single node: This set of experiments consists of two settings: `SETTING A`, which does not use APS partitions (and hence it cannot leverage partition inheritance but only priority inheritance, if enabled), adopting fixed-priority scheduling without reservation, and `SETTING B`, which instead uses APS.

`SETTING A` executes four client threads and one server thread on the same core. Each client thread executes for 50% of its execution time before a request is sent to the server. All clients request the same service with a WCET of 10 ms and always run for their WCET. A complete description of the setting is provided in the lower part of Fig. 2a. Client and server parameters in the tuple are described in the following order: period, WCET, offset, nominal priority and required/provided service. As a server does not have its own period or offset we use '$-$' to indicate this at the respective places. The initial offsets of client threads are selected such that the server s_1 handles the initial request of c_1 before c_2, c_3 and c_4 can each send a request to the server.

Listing 1 and 2 show an excerpt of the execution trace of c_2 and s_1, respectively, for the case in which inheritance is disabled. From the trace of c_2 it can be observed that the client thread transitions to `READY` at its release at $t = 6$ ms, confirming **R1**. After the initial execution phase, c_2 sends a request to s_1 and transitions to the state `SEND_BLOCKED`, confirming **R2**. From the server trace, it can be seen that s_1 transitions to `RECEIVE_BLOCKED` at $t = 74.5$ ms when no more pending messages exist. The server becomes `RUNNING` again after c_1 sends its second request (note that the trace does not include an explicit entry for `READY` as s_1 directly starts to execute). This confirms **R3** and **R4**. The trace of c_2 further shows that the thread's state transitions from `SEND_BLOCKED` to `REPLY_BLOCKED` at $t = 59.6$ ms, which coincides with the respective execution start of s_1 to handle the request. At



(a)



(b)

Figure 2: `SETTING A` *without* priority inheritance and *without* APS (a) and *with* priority inheritance and *without* APS (b). Execution trace of the first 80 ms. The thread parameters of `SETTING A` are shown in the bottom party of (a) and the priority trace of s_1 is shown in the bottom part of (b).

$t = 69.5$ ms the server completes the request and c_2 transitions to `RUNNING`, confirming **R5**.

Listing 1: Partial trace of c_2.

```
 6.0015 ms READY         CPU: 1 Prio: 253
 6.0052 ms RUNNING       CPU: 3 Prio: 253
10.9591 ms SEND_BLOCKED  CPU: 3 Prio: 253
59.6297 ms REPLY_BLOCKED CPU: 3 Prio: 253
69.5848 ms RUNNING       CPU: 3 Prio: 253
```

Listing 2: Partial trace of s_1.

```
 54.6692 ms READY           CPU: 3 Prio: 252
 59.6271 ms RUNNING         CPU: 3 Prio: 252
 69.5835 ms READY           CPU: 3 Prio: 252
 74.5402 ms RUNNING         CPU: 3 Prio: 252
 74.5426 ms RECEIVE_BLOCKED CPU: 3 Prio: 252
204.9591 ms RUNNING         CPU: 3 Prio: 252
```

Fig. 2a depicts the same execution trace where inheritance is *not* enabled. At $t = 30$ ms s_1 finishes serving the request of c_1. There are three outstanding requests in the queue of the server that arrived in the sequence c_2, c_3, and c_4. The server first handles the request by c_3, then the request by c_4, and finally the request of c_2. Therefore, **R7** is confirmed, and services are handled in priority order by the server. **R6** can further be confirmed as the initial request of c_1 is served to completion before other requests are handled in priority order. This is a direct consequence of the server implementation (see Algorithm 1).

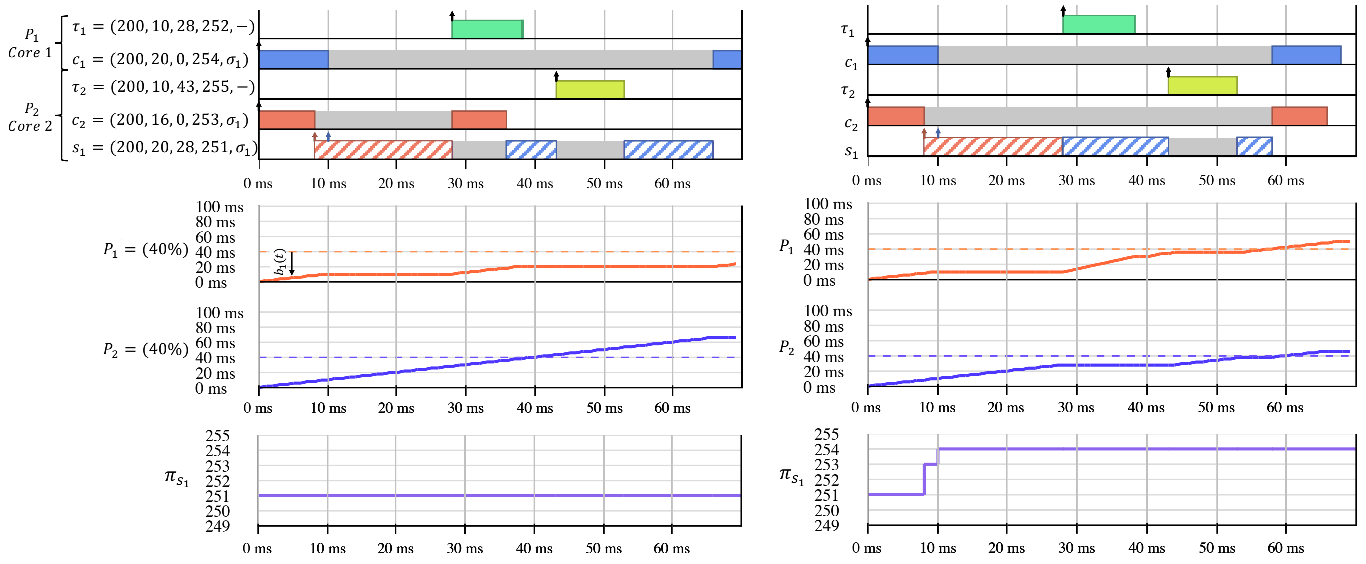
Fig. 2b depicts the same system, but inheritance *is* enabled. The priority of s_1 is shown in the lower part of Fig. 2b. s_1 starts to execute at its nominal priority 252. At $t = 5$ ms s_1 starts to handle the request of c_1. Consequently, s_1 inherits the priority of c_1 and executes at a priority of 250. At $t = 11$ ms, c_2 sends a request to the server, followed by a request from c_3 at $t = 17$ ms. We can observe that the server priority is following accordingly, to 253 at $t = 11$ ms and to 255 at $t = 17$ ms. This confirms the first part of **R8**. A difference between Fig. 2a and Fig. 2b can be observed at $t = 35$ ms. s_1 executes at the highest priority and completes the request of c_1. The current priority of s_1 is 255, and only one request of c_2 is pending (with priority 253). s_1 continues executing at priority 255 to call `MsgReceive()` and retrieve the pending request by c_2, and its priority changes to 253 before the server is preempted by c_4 (confirming the second part of **R8**). Thus, the order of requests that are handled by the server is different compared to the case without priority inheritance.

`SETTING B` is used to examine how APS partitions are handled in conjunction with the SyncMP mechanism. The system comprises five threads in total. Two threads act as client threads (c_1 and c_2), one server thread (s_1), and two ordinary threads (τ_1 and τ_2). As before, each client executes for half of its nominal execution time before it places a request to the server. Thread parameters are listed next to the execution trace in Fig. 3. Please note that the tracing tool of QNX does not show the current budget of each partition, but the cumulative time that threads of a respective partition executed within the last scheduling window $[t - W, t)$: the corresponding budget can be read as shown for P_1 in Fig. 3(a). τ_1 and c_1 are assigned to partition P_1 and τ_2, c_2 and s_1 are assigned to partition P_2. Each APS partition has a nominal budget of 40%. All threads that are mapped to P_1 are allocated to core 1, and all threads that are mapped to P_2 are allocated to core 2. Fig. 3 depicts the recorded execution trace (top), APS trace (middle), and priority trace (bottom) for the case with (a) and without (b) inheritance enabled.

In the case without inheritance (Fig. 3a), we can observe that the SyncMP mechanism is not affected by the APS scheduler. Furthermore, each APS partition consumes budget when a respective thread executes. For P_2 this means the server exceeds its nominal budget at $t = 40$ ms, but since no other thread is assigned to the same core P_2 can consume the idle time. We can also observe that a thread that preempts a server, while the server serves a request using an inherited APS partition, executes in its nominal partition (τ_2 between $t = 43$ ms and $t = 53$ ms). From these observations, we can conclude that **R9-A** holds for the case that client and server are mapped to the same node.

2) Experiments on a distributed system: This section investigates the SyncMP properties when client and server are allocated on two different nodes. In our experimental setup, both nodes are identical to the one used for the single-node experiments and connected via Ethernet.

`SETTING C` executes two client threads c_1 and c_2 and one server thread s_1 on two different nodes connected via QNET. c_1 is allocated to Node 1 and a partition P_1. c_2 and s_1 are



(a) Inheritance Disabled

(b) Inheritance Enabled

Figure 3: (top) Execution trace of SETTING B in the first 70 ms. (middle) Partition budget usage of P_1 (red), P_2 (blue). (bottom) Priority of server s_1. The thread parameters are described in (a) in the following order: period, WCET, offset, nominal priority, required/provided service. The assigned partition and core are indicated for the thread groups. Each partition is described by its budget B_k, and W =100 ms. The current budget $b_k(t)$ is illustrated for P_1 in (a).

allocated to Node 2 and a partition P_2. Inheritance is enabled for the communication channel. Thread parameters can be found in Fig. 4. Fig. 4 shows the recorded trace of Node 1 and Node 2. Note that timestamps are not synchronized and exhibit an about 5 ms offset due to independent trace collection. The key findings from this experiment are:

(1) Priority inheritance works across node boundaries. This can be seen as π_{s_1} inherits the priority of c_1 and c_2 at $t = 200$ ms and $t = 220$ ms respectively.

(2) Partition inheritance behaves differently across node boundaries. While requests by c_2 are served by consuming budget of P_2, requests by c_1 are not served by consuming budget of P_2. In fact, requests of c_1 are neither consuming the budget of $P1$ or $P2$ but instead consume budget from the (default) system partition. This observation corroborates the validity of **R9-B**.

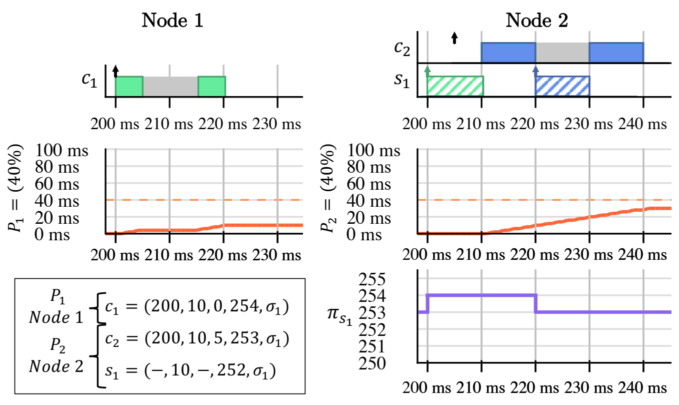


Figure 4: Execution trace of SETTING C on Node 1 and Node 2 (note: timestamps on different nodes are not synchronized).

E. Discussion

In this section, we formalized the behavior of the QNX OS SyncMP mechanism through a set of rules, and we validated them with a series of experiments on a real platform, paving the way for a sound timing analysis. We observe scenarios that make the timing analysis challenging. This is the case when two threads are consuming the budget of an APS partition at the same time. This is shown in Fig. 3b, between $t = 28$ and $t = 38$, where the budget in P_1 is consumed by τ_1 and s_1.

Furthermore, experiments with the client and the server distributed over different nodes highlight the importance of performing them, uncovering undocumented behaviors (**R9-B**) with substantial effects on the timing behavior. One possible explanation for the observed APS inheritance across nodes is the lack of platform information across different nodes (CPU frequency etc.), while the APS mechanism keeps track of budgets in clock ticks [8]. Exchanging APS information would further likely lead to increased communication bandwidth requirements.

IV. Response-Time Analysis

The problem we address in this section is the following: *how to bound the WCRT of a client thread that uses services provided by a server in a QNX-based system?* We start discussing the case in which the priority and partition inheritance mechanism is not active. Before proceeding, we recall the needed background for analyzing self-suspending tasks.

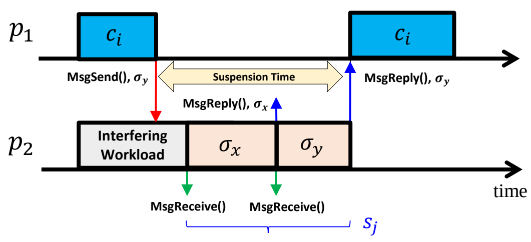


Figure 5: Client-server interaction as a self-suspension.

A. Background on self-suspending tasks

A client thread c_i can be modeled as a self-suspending task [9], by noting that every time c_i uses `MsgSend()` to request a service (rule **R2**), it self-suspends until the service is completed and the corresponding server calls `MsgReply()` system call (rule **R5**). This is shown in Fig. 5.

To introduce the needed background for bounding the WCRT of a self-suspending task (SS-task), we consider a set $\Gamma_i^{\text{ss}} = \{\tau_1^{\text{ss}}, \dots, \tau_n^{\text{ss}}\}$ related to a partition P_k. Each SS-task is characterized by a minimum inter-arrival time T_i, a constrained deadline $D_i \leq T_i$, a WCET e_i, and a suspension time bound S_i, under fixed-priority partitioned scheduling.

Furthermore, let $\text{hep}_z^{\text{ss}}(\tau_i^{\text{ss}})$ be the set of SS-tasks with a higher or equal priority of τ_i^{ss} on the same core p_x, excluding τ_i^{ss} itself. The WCRT R_i of a task τ_i^{ss} can be bounded with the least positive solution of the following equation:

$$sbf_k(R_i) \geq e_i + S_i + \sum_{\tau_j^{\text{ss}} \in \text{hep}_z^{\text{ss}}(\tau_i^{\text{ss}})} \left\lceil \frac{R_i + \overline{R}_j - e_j}{T_j} \right\rceil \cdot e_j \quad (1)$$

by modeling the self-suspension of the task under analysis as execution time and self suspensions of interfering tasks as jitter [9]. In Eq. (1), which computes the response time bound R_i, it is assumed to use a pre-existing WCRT bound $\overline{R}_j$ for τ_j^{ss}, thus introducing a cyclic dependency. The dependency can be broken by assuming that all jobs are killed after the deadline: this allows initially setting the WCRT bounds $\overline{R}_j$ to the deadline D_j. In this way, if threads are schedulable, no jobs are killed, making the assumption not necessary (more details in [16]). Iterative improvements are possible by running the analysis multiple times for all tasks, each time obtaining new (smaller) bounds for R_i leveraging $\overline{R}_j$ (see Eq. (1)), updating the values of each $\overline{R}_i$ with the newly computed R_i, and re-iterating the process until no improvement is achieved between two consecutive runs of the analysis [16]. Finally, note that the equation of Section 4.2.3 in [9] has been extended in Eq. (1) to consider a more general supply-bound function $sbf_k(\Delta)$, while [9] considered a core supply equal to the total core capacity. The extension is safe as long as the corresponding partition does not deplete the budget when it becomes idle (e.g., as it occurs for a polling reservation [12]). However, this is not the case for the APS partition reservation algorithm [6], which ensures each suspending thread that the remaining budget after being awakened from the suspension will be no less than the one it had before suspending if no interfering workload in the same partition consumed it in the meantime. Our analysis, presented in the following, remains compatible with the case of a fully-dedicated core, which can be modeled with each core having a single partition allocated on it, with a supply-bound function $sbf_k(\Delta) = \Delta,\ \forall p_k \in \mathcal{U}$.

B. Baseline Response-Time Analysis (C-S)

Deriving a bound on the suspension time. We start focusing on the `C-S` scenario, which consists in the baseline client-server mechanism with no inheritance enabled. By rule **R2**, a client task self-suspends for waiting for its service request to be served. Therefore, to build a response-time analysis framework to guarantee the schedulability of client threads, we first bound the time spent waiting by service requests to servers, i.e., the suspension time, in the SS-task theory jargon.

For each service σ_x, let $R_{x,i}^{\text{sc}}$ be a bound on its worst-case response time when the service request is issued by c_i, i.e., the longest time span from when a service request for σ_x is received by its server $s_h = \mathcal{Q}^{-1}(\sigma_x)$ to when it completes (i.e., when `MsgReply()` is used to reply, see **R5**).

Therefore, the overall self-suspending time of a client c_i is bounded by the sum of the WCRT bounds all the services multiplied by $n_{i,x}^{\text{S}}$, the number of times c_i calls σ_x, plus the WCCD required for c_i to communicate with s_h for service σ_x:

$$S_i \leq \sum_{\sigma_x \in \mathcal{Q}} \left(n_{x,i}^{\text{S}} \cdot (R_{x,i}^{\text{sc}} + \delta_{i,h}^{x} + \delta_{h,i}^{x})\right) \quad (2)$$

Next, we bound the WCRT $R_{x,i}^{\text{sc}}$ of an arbitrary service σ_x in setting `C-S`. To this end, let $s_h = \mathcal{Q}^{-1}(\sigma_x)$ be the server that manages σ_x.

To begin, we aim at deriving an analysis that is as flexible as possible, allowing the co-existence of multiple clients and servers in the same partition (which can be helpful in some cases since APS limits the maximum number of partitions on one node to 32 [7]).

Hence, given an arbitrary service σ_x under analysis handled by a server s_h allocated to a partition P_k, we classify three different sources of interference:

- The interference due to other clients allocated to the same APS partition P_k, bounded by $I_h^{\text{cl}}(\Delta)$.
- The interference caused by any other server than s_h in P_k, bounded by $I_h^{\text{oth-sr}}(\Delta)$.
- The interference due to the processing of other service requests in the same server s_h, bounded by $I_{h,x}^{\text{sm-sr}}(\Delta)$.

Based on these sources of interference, Lemma 1 presents a bound on the WCRT of a service σ_x issued by a client c_i.

Lemma 1. *Consider an arbitrary instance σ'_x of a service σ_x released at time A and issued by a client c_i. If S^* is the least positive solution of the following inequality*

$$sbf_k(S^*) \geq \epsilon + I_h^{cl}(S^*) + I_h^{oth\text{-}sr}(S^*) + I_{h,x}^{sm\text{-}sr}(S^*) \quad (3)$$

then σ'_x starts running at most at $A + S^*$. If R^* is the least positive solution of the following inequality

$$sbf_k(R^*) \geq \epsilon + I_h^{cl}(R^*) + I_h^{oth\text{-}sr}(R^*) + I_{h,x}^{sm\text{-}sr}(S^*) + \omega_{x,i} \quad (4)$$

where then σ'_x completes at most at $A + R^*$ and $R^{sc}_{x,i} = R^*$ is a bound on the response time of σ_x issued by c_i.

Proof. By definition $I^{\text{cl}}_h(S^*)$, $I^{\text{oth-sr}}_h(S^*)$, and $I^{\text{sm-sr}}_{h,x}(S^*)$ bound the interference due to clients, services running in other servers in the same APS partition, and services handled by the same server in the interval $[A, A + S^*)$. Since S^* fulfills Eq. (3), the supply provided by partition P_k exceeds the demand by at least ϵ. Therefore, since σ'_x is released at A, σ'_x starts being handled by s_h at most at $A + S^*$.

To prove Eq. (4), note that, internally to server s_h, σ'_x is handled in a non-preemptive fashion. It follows that in $[A + S^*, A + R^*)$, σ'_x cannot suffer interference from other services handled by the same server. Hence, the overall interference due to other service requests in $[A, A+R^*)$, is bounded by $I^{\text{sm-sr}}_{h,x}(S^*)$. Conversely, interference due to other client threads and requests handled in other servers can interfere also within $[A + S^*, A + R^*)$, and the corresponding interference components are bounded by $I^{\text{cl}}_h(R^*)$ and $I^{\text{oth-sr}}_h(R^*)$. Since by assumption $R^{\text{sc}}_{x,i} = R^*$ satisfies Eq. (4), the supply $sbf_k(R^*)$ provided by P_k exceeds the demand (which includes the aforementioned interference components and the WCST $\omega_{x,i}$) by at least ϵ. It follows that σ'_x completes within $[A, A+R^*)$, and R^* is a valid response-time bound for σ'_x. $\square$

Thanks to Lemma 1, we are able to bound the WCRT $R^{\text{sc}}_{x,i}$ of a service provided that the delay components $I^{\text{cl}}_h(\Delta)$, $I^{\text{oth-sr}}_h(\Delta)$, and $I^{\text{sm-sr}}_{h,x}(\Delta)$ are know. Next, we provide bounds for such components.

Similarly to classical jitter-based analysis for self-suspending tasks (recalled in Section IV-A), the interference due to clients $I^{\text{cl}}_h(\Delta)$ is bounded by the sum of the individual interference contributions by higher-or-equal-priority clients in the same partition P_k of the server s_h handling σ_x. Hence, it holds,

$$I^{\text{cl}}_h(\Delta) \triangleq \sum_{c_j \in \text{hep}^{\text{cl}}_k(s_h)} \left\lceil \frac{\Delta + \overline{R}_j - e_j}{T_j} \right\rceil \cdot e_j. \qquad (5)$$

We omit a formal proof to show Eq. (5) to be sound as it is analogous to standard results for self-suspending tasks [9]. As discussed in Section IV-A, Eq. (5) requires a pre-existing bound on the response time of high-priority clients $c_j \in \text{hep}^{\text{cl}}_k(s_h)$, which can initially be upper-bounded with its deadline D_j, and possibly refined in an iterative fashion as discussed in Section IV-A. This technique, well-known in literature (e.g., see [9, 16]–[19]), is used multiple times in the following: the corresponding pre-existing WCRT bounds are denoted with the symbol $\overline{R}$ to distinguish from those are under derivation (denoted by R).

Next, we define $I^{\text{oth-sr}}_h(\Delta)$.

Lemma 2. *In setting* C-S*, it holds*

$$I^{oth\text{-}sr}_h(\Delta) \triangleq \sum_{s_a \in hep^{sr}_k(s_h)} \sum_{\sigma_s \in \mathcal{Q}(s_a)} \sum_{c_e \in C \setminus c_i} \left\lceil \frac{\Delta + \overline{R}_e}{T_e} \right\rceil \cdot n^S_{s,e} \cdot \omega_{s,e}. \qquad (6)$$

Proof. First, note that any other server with a priority higher than or equal to s_h can interfere with s_h (s_h is excluded by definition of $I^{\text{oth-sr}}_h(\Delta)$). These servers are contained into the set $\text{hep}^{\text{sr}}_k(s_h)$. For each interfering server s_a, each service σ_s handled by that server (i.e., in set $\mathcal{Q}(s_a)$) can interfere with σ_x under analysis. Each service request is triggered at most $n^{\text{S}}_{e,s}$ times by each instance of each client $c_e \in C \setminus c_i$. Note that the client c_i issuing the request for σ_x under analysis can be excluded from the set of clients issuing interfering requests because the QNX SyncMP is *synchronous* (rules **R2-R5**) and hence c_i has at most a pending service request at the same time. Knowing that each service σ_s can take at most $\omega_{s,e}$ time units, it remains to bound the maximum number of service requests issued by each client $c_e \in C \setminus c_i$ that may contribute to interference in an arbitrary interval $[t, t+\Delta)$. Such requests are determined by client's jobs, which are released according to a minimum inter-arrival time T_e. By definition of WCRT bound, service requests issued by clients job released before $t - \overline{R}_e$ must have completed at t, and therefore they cannot contribute to the interference in $[t, t+\Delta)$. Therefore, only jobs released in $[t - \overline{R}_e, t+\Delta)$ may contribute. The lemma follows by noting that at most $\left\lceil \frac{\Delta + \overline{R}_e}{T_e} \right\rceil \cdot n^{\text{S}}_{e,s}$ service requests then be issued in $[t - \overline{R}_e, t + \Delta)$. $\square$

Lemma 3 bounds the interference due to other services handled by the same server.

Lemma 3. *In setting* C-S*, it holds*

$$I^{sm\text{-}sr}_{h,x}(\Delta) \triangleq \sum_{\sigma_s \in \mathcal{Q}(s_h)} \sum_{c_e \in hep^{cl}(c_i)} \left\lceil \frac{\Delta + \overline{R}_e}{T_e} \right\rceil \cdot n^S_{s,e} \cdot \omega_{s,e} + B^{sm\text{-}sr}_{h,x}, \qquad (7)$$

where $B^{sm\text{-}sr}_{h,x} = \max_{\sigma_l \in L, c_b \in lp^{cl}(c_i)} \omega_{b,l}$, with $L = \{\sigma_l \in \mathcal{Q}(s_h) : \exists c_b \in lp^{cl}(c_i) \wedge n^S_{b,l} > 0\}$.

Proof. By rule **R7**, any service request issued by a client with priority higher than or equal to c_i in any partition (set $\text{hep}^{\text{cl}}(c_i)$) can interfere with σ_x in analysis. Analogously to Lemma 2, each of such service instance interfere for up to $\left\lceil \frac{\Delta + \overline{R}_e}{T_e} \right\rceil \cdot n^{\text{S}}_{e,s} \cdot \omega_s$ time units. By rule **R6**, each service runs to completion once started. Therefore, any low-priority service started just before σ_x is issued can block σ_x itself, for at most the duration of a single service. Low-priority blocking is then bounded by the longest duration of a service request from a low-priority client, i.e., by $B^{\text{sm-sr}}_{h,x}$, proving the bound. $\square$

Bounding the WCRT of clients. The previous lemmas allow to bound the suspension time. Such a bound is used next in Lemma 4 to bound the WCRT of a client.

Lemma 4. *Consider an arbitrary instance of c_i released at time A. If R_i is the least positive solution of the following inequality*

$$sbf_k(R_i) \geq \epsilon + e_i + S_i + I^{cl}_i(R_i) + I^{oth\text{-}sr}_i(R_i) \qquad (8)$$

then c_i completes no later than $A + R_i$ and R_i is a bound on the response time of c_i in setting C-S*.*

Proof. A client c_i can be delayed either by: **(i)** interference due to other clients and servers running in the same partition P_k, **(ii)** lack of supply due to P_k, and **(iii)** suspension time due to the synchronous offloading to servers (following the *suspension-oblivious* approach to model the suspension of the interfered thread [9], see Section IV-A). Delays due to (i) are accounted in $I_i^{\text{cl}}(R_i)$ and $I_i^{\text{oth-sr}}(R_i)$ delays due to (ii) are accounted in the supply-bound function $sbf_k(R_i)$, and (iii) is accounted for in S_i, which can be bounded with the results of Section IV-B. Since by assumption R_i satisfies Eq. (8), the supply $sbf_k(R_i)$ provided by P_k exceeds the demand (which includes the interference plus at most e_i units to execute c_i) by at least ϵ. It follows that σ'_x completes no later than $A+R_i$, and R_i is a valid response-time bound for c_i. □

The definitions of $I_i^{\text{cl}}(\Delta)$ and $I_i^{\text{oth-sr}}(\Delta)$ are analogous to those presented in Section IV-B (i.e., Eq. (5) and Eq. (6)) but specialized by using the sets $\text{hep}_k^{\text{cl}}(c_i)$ and $\text{hep}_k^{\text{sr}}(c_i)$ in place of $\text{hep}_k^{\text{cl}}(s_h)$ and $\text{hep}_k^{\text{sr}}(s_h)$ to denote the sets of all clients and servers with higher or equal priority than c_i, respectively, and without excluding c_i from C in the sum of Eq. (6).

Analysis and modeling alternatives. In this paper, we adopt the more flexible dynamic self-suspending model [9], modeling the suspension of the interfered thread as computation (suspension-oblivious) and the suspension of interfering threads as released jitter. From the modeling side, we deem it more accessible for an application designer to know only the *overall* WCET of a client and a suspension-oblivious approach. Conversely, adopting the segmented self-suspending model requires knowing the WCET bounds of each code chunk between two consecutive service calls, which can be hard to obtain. Nevertheless, when this information is available, a SPLIT analysis [9] approach could be adopted, and precision can possibly be improved. However, none of the approaches dominates the other, as SPLIT would incur additional accounting (for each segment) of the supply-bound function blackout time [6]. In contrast, the suspension-oblivious approach adopted in this paper may be more pessimistic when the client and the server are in the same core, as the same high-priority interference could be counted twice, both in the client and the server interference. The combination of the proposed method with SPLIT and the derivation of ad-hoc methods (e.g., using MILP formulations) to reduce the pessimism will be addressed in future work.

C. Response-Time Analysis with Inheritance (LOCAL-I)

Next, we analyze the `LOCAL-I` setting, which occurs when inheritance is enabled, and pairs of clients and servers are on the same node. This setting is much harder to analyze in the general case. We identify two main analysis challenges and propose solutions that allow keeping the analysis simple.

C1 - Multi-supply over time. A pending service request instance σ'_x under analysis can be interfered with by high-priority requests from different clients, which can be allocated to different APS partitions with different configurations and hence with a different supply-bound function $sbf_k(\Delta)$. Thus, a general analysis should derive the worst-possible sequence of interfering requests to σ'_x and the state of the supply functions of the corresponding (potentially different) client partitions when such service requests produce interference in order to provide the lowest possible combination of supply provided to the server.

C2 - Double-rate use of the budget. If the client's partition P_k includes multiple threads and another partition P_y starts consuming P_k's budget due to the partition inheritance mechanisms, P_k and P_y can consume the budget of P_k in parallel, at a double rate, making it hard to lower-bound the partition supply (see Fig. 3b between $t = 28$ and $t = 38$, where the budget in P_1 is consumed by c_1 and s_1).

Due to its complexity, we leave the general case to future work, and we target a more accessible yet practical case.

Therefore, we make the following simplifying assumptions:
A1. Each server handles requests from only one client.
A2. No other thread runs in the same partition of a client.
A3. No other thread runs in the same partition of a server.

This setting is representative of many real cases where it is desirable to free clients and servers from interference due to other workloads, allocating them to dedicated partitions while still allowing to serve multiple clients by splitting the server code into multiple server threads.

Assumptions **A1-A3** are largely beneficial from a predictability perspective. Thanks to **A2** and **R2-R5**, at most one thread can consume the budget of a client's partition P_k at a time, thus avoiding the supply to be consumed at a double rate (**C1**). This allows using the existing bound for $sbf_k(\Delta)$ that leverages this assumption [6]. With **A1**, the server partition inherits the budget of a single partition, i.e., the one of its client, thus avoiding the problem of combining the supply function of the server partition with multiple supply functions of different clients (**C2**). Finally, with **A3**, the server partition can be configured with zero budget, thus also avoiding combining its supply function with the one of the corresponding client. Lemma 5 bounds the response time under **A1-A3**.

Lemma 5. *Consider an arbitrary instance of c_i released at time A. If **A1**, **A2**, and **A3** hold, and R_i is the least positive solution of the following inequality*

$$sbf_k(R_i) \geq e_i + \sum_{\sigma_x \in \mathcal{Q}} (n_{x,i}^{S} \cdot (\omega_{x,i} + \delta_{i,j}^{x} + \delta_{j,i}^{x})) \qquad (9)$$

then c_i completes no later than $A + R_i$ and R_i is a bound on the response time of c_i in setting `LOCAL-I`.

Proof. Due to **A1**, the server thread $s_h = \mathcal{Q}^{-1}(\sigma_x)$ handles requests only from the client c_i under analysis. Due to rules **R2-R5**, the considered IPC mechanism is synchronous. Due to **A3**, no other workload run in the partition of the server thread $s_h = \mathcal{Q}^{-1}(\sigma_x)$. Therefore, a service request $\sigma_x \in \mathcal{Q}$ with $n_{x,i}^{\text{S}} > 0$ is immediately ready to be served in the server when received, if the client's partition has current budget $b_k(t) > 0$. Thanks to **A2**, no other workload can run in parallel in the client's partition P_k, thus making $sbf_k(\Delta)$

(as defined in [6]) a valid abstraction for the service supply provided by P_k (i.e., challenge **C2** is avoided). The lemma then follows by noting that the resulting scheduling behavior is compatible with the one of a self-suspending task with no interference (i.e., executed in isolation), which can be analyzed by accounting for the suspension time as computation [9], with $S_i = \sum_{\sigma_x \in \mathcal{Q}} (n^{\mathrm{S}}_{x,i} \cdot (\omega_{x,i} + \delta^x_{i,j} + \delta^x_{j,i}))$. $\square$

D. Response-Time Analysis with Inheritance (DISTR-I)

Finally, we analyze the setting `DISTR-I`, which occurs when the client c_i and the server s_h are in two different nodes, and they communicate through `QNET` (see Section III-D). `DISTR-I` results in having the expected inheritance of priorities, but requests are managed using the system's partition budget (rule **R9-B**). This introduces a third analysis challenge: **C3 - Derivation of the *residual* supply-bound function.** In APS, the system partition is the parent partition from which all partitions are subtracted when created. Therefore, to bound the budget that could be used to run the service requests directed to a server, it is necessary to bound the *residual* supply-bound function [20] $sbf_{sys,p_q}(\Delta)$.

In principle, this is possible by bounding the demand due to all non-system partitions allocated on a specific core and subtracting it from the overall processing time provided by the core. We leave the derivation of the residual supply-bound function under APS to future research while we focus on a simpler yet practical setup. To simplify the analysis, in this case we assume that:

A4. Servers are allocated in dedicated cores where all threads are only assigned to the system partition. These cores host no partitions other than the system partition.

Under this assumption, the residual supply-bound function for a core p_q equals the overall processing time provided by the core, i.e., $sbf_{sys,p_q}(\Delta) = \Delta$. Instead, under `DISTR-I`, challenges **C1** and **C2** are not present because they are caused by the inheritance of the budget from the client partition. Also, note that the solution used for the `LOCAL-I` setup does not work in this case since it exploits the fact that under `LOCAL-I` the server uses the budget of the client partition. To extend the baseline analysis of Scenario `C-S` to work in Scenario `DISTR-I`, note that this affects the suspension time bound of Section IV-B as follows. First, the suspension time bound needs to consider the system partition under assumptions **A4**, with $sbf_{sys,p_q}(\Delta) = \Delta$. Second, the suspension bound of Scenario `C-S` needs to be modified to account for priority inheritance (rule **R8**). Priority inheritance is an inter-thread phenomenon. Hence, the interference due to other service requests in the same server (term $I^{\text{sm-sr}}_{h,x}(\Delta)$) is analogous to Lemma 3. Differently, the interference due to other threads need to be adapted to consider the inheritance due to rule **R8**.

To this end, we first specialize the definition of $I^{\text{cl}}_h(\Delta)$ using the set $\text{hep}^{\text{cl}}_{sys,p_q}(c_i)$ in place of $\text{hep}^{\text{cl}}_{sys,p_q}(s_h)$ to denote the sets of all clients with higher or equal priority than c_i since due to **R8** the set of clients that can interfere with σ_x under analysis depends on the priority of c_i (and not of s_h, as in setting `C-S`).

Lemma 6 defines $I^{\text{oth-sr}}_h(\Delta)$ for the `DISTR-I` setting.

Lemma 6. *In setting `DISTR-I`, under **A4**, it holds $I^{\text{oth-sr}}_h(\Delta) \triangleq B^{\text{inh}}_{h,x} + I^{\text{hep}}_h(\Delta)$, where*

$$I^{hep}_h(\Delta) = \sum_{s_a \in S_{sys,p_q} \setminus s_h} \sum_{\sigma_s \in \mathcal{Q}(s_a)} \sum_{c_e \in hep^{cl}(c_i)} \left\lceil \frac{\Delta + \overline{R}_e}{T_e} \right\rceil n^S_{s,e} \omega_{s,e}, \tag{10}$$

and

$$B^{inh}_{h,x} = \sum_{s_a \in S_{sys,p_q} \setminus s_h} \max_{\sigma_l \in L, c_b \in lp^{cl}(c_i)} \omega_{b,l}, \tag{11}$$

with $L = \{\sigma_l \in \mathcal{Q}(s_a) : \exists c_b \in lp^{cl}(c_i) \wedge n^S_{b,l} > 0\}$.

Proof. Consider a service request σ_x under analysis issued by a client c_i and managed by a server $s_h = \mathcal{Q}^{-1}(\sigma_x)$. $I^{\text{oth-sr}}_h(\Delta)$ consists of two components: **(i)** interference due to service requests from higher-or-equal priority clients $c_e \in \text{hep}^{\text{cl}}(c_i)$ and **(ii)** blocking due to other servers that boost their priority. Consider first (i). By **R8**, since each server inherits the priority of the client issuing a request, in principle any server $s_a \in S_{sys,p_q} \setminus s_h$ in the system partition of the same core can interfere with s_h (s_h excluded). For each interfering server s_a, due to **R8**, only services σ_s issued by clients c_e with higher or equal priority can interfere, irrespective to the partition, i.e., those from $\text{hep}^{\text{cl}}(c_i)$. Therefore, similar to Lemma 2, (i) is bounded by $I^{\text{hep}}_h(\Delta)$. Consider now (ii). We first show that each server $s_a \in S_{sys,p_q} \setminus s_h$ can block at most once σ_x. When server s_a starts processing a request from a client $c_b \in \text{lp}^{\text{cl}}(c_i)$, it may receive a service request from a client c_x with priority higher than π_i. By the second part of **R8**, then s_a rises its priority π_x, thus causing blocking to the service request under analysis. Still due to **R8**, once s_a calls again `MsgReceive()`, it either **(a)** lowers its priority, thus being preempted by s_h, or **(b)** starts servicing a higher-priority request than the one under analysis. In both cases, σ_x cannot be blocked again. Finally, each server $s_a \in S_{sys,p_q} \setminus s_h$ cannot contribute to blocking for more than the length of the longest service requested by a client $c_b \in \text{lp}^{\text{cl}}(c_i)$ that s_a manages. Hence, (ii) is bounded by $B^{\text{inh}}_{h,x}$. The lemma then follows. $\square$

Finally, note that whenever a client is allocated to the same core of one or more servers, the response-time bound of Lemma 4 needs to be updated with a blocking term, similar to Eq. (11).

V. Evaluation

This section presents the results of three evaluation studies. The first one evaluates the communication delay $\delta^x_{i,j}$ in both the local and distributed setting. The second study targets a synthetic case where we compare the WCRT provided by the analysis with measurements obtained by running the application on an actual platform running QNX. All experiments on the QNX platform use the same setup that is used for the validation experiments in Sec.III-D. Finally, we evaluate the analysis on a realistic-case study based on the WATERS 2019 Challenge by Bosch [10] for both local and distributed setups.

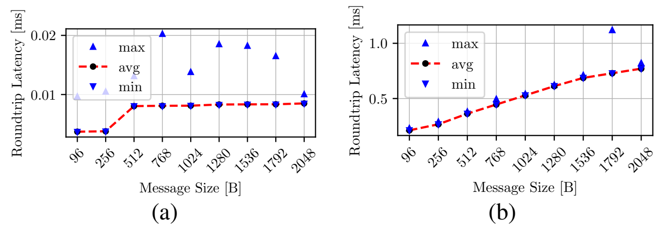


Figure 6: Round-trip latency of the IPC communication for a varying length of request and reply message on the same (a) and between two nodes (b).

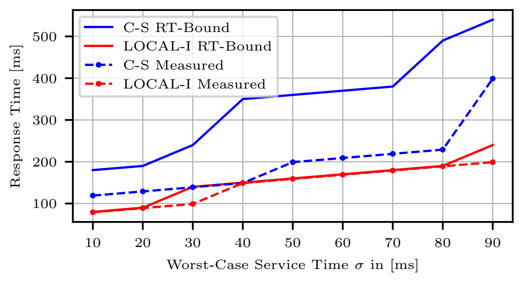


Figure 7: WCRT bounds vs. measurement results.

A. *Evaluation of Communication Latency.*

The experimental setup used is the same as described in Section III for the validation experiments, in which we used one client and one server. The server does not perform any computation and replies directly after receiving the message. In this experiment, we recorded the latency required to perform and return from a call to `msg_send()`, issued by the client. The measured latency thus includes both the time required to reach the server from the client and to come back from the server to the client. A varying message size (same value for request and response message) is used in the experiment for messages in the range $[96\,\mathrm{B}, 2048\,\mathrm{B}]$, and 1000 measurements are performed for each configuration.

Fig. 6 shows the results for the case where client and server are allocated on the same node (Fig. 6a) and on different nodes (Fig. 6b). In the local communication case, for the observed range of message sizes, the communication delay remains almost constant for messages smaller than $256\,\mathrm{B}$ and again for messages with a size larger than $256\,\mathrm{B}$. This jump in latency is likely caused by data-size-dependent implementations to copy data within the kernel (e.g., page flipping instead of the usual copy operations), as indicated in the QNX manual [5]. The observed latencies in the distributed case increase more distinctly with varying message size. In addition to the pure communication delay, QNET protocol overheads are observed.

B. *Comparing Measurement Against Analysis Results.*

Then, in the second evaluation study, we compared the analytical WCRT bound on the response time of a client against the largest observed value within several experiment runs (50 instances per data-point) of $10\,\mathrm{s}$ on the QNX platform, with and without inheritance enabled. A client c with $T = 200\,\mathrm{ms}$, $e = 20\,\mathrm{ms}$ is assigned to a partition P_1 on a core A, with 60% of the overall budget and requires a service σ. A server that serves σ is allocated to a separate partition P_2 on a core B with 40% of the overall budget. The interfering workload is allocated to both cores in separate partitions, consuming the remaining budget, which would have otherwise been used by P_1 and P_2 leveraging APS's reclaiming. Priorities are assigned such that the interfering workload on core A has a higher priority than the client; the client has a higher priority than the interfering workload on core B, and the server has the lowest priority. Measurements are obtained for a varying worst-case service time of σ in the interval $[10, 100]\,\mathrm{ms}$ in steps of $10\,\mathrm{ms}$.

Fig. 7 shows that the measurement results for the case with inheritance are very close to the computed bounds, demonstrating the tightness of our approach for this setting.

C. *Automotive Case Study.*

The case study investigates an application based on a next-generation automotive application, originally presented at WATERS'19 [10]. The application realizes an end-to-end autonomous driving application as shown in Fig. 8. For a detailed description of the case study we refer to [19, 21].

For the evaluation, the focus is put on the threads *Detection* and *Localization*. In the original model, both threads offload parts of their computation to accelerators. We adapted the case study to realize this by the QNX's SyncMP mechanism. Thus, *Detection* requires the service σ_1 with a WCST $\omega_2 = 99.8\,\mathrm{ms}$ and *Localization* requires the service σ_2 with a WCST $\omega_1 = 99.2\,\mathrm{ms}$. For each service, a server thread is added to the system and serves σ_1 and σ_2, respectively. Each client and server is allocated to an APS partition. If not otherwise stated, the client thread partitions and server partitions have a budget of 50% and 60% of the overall budget of their cores, respectively. Priorities are assigned based on a rate monotonic order, and the servers are assigned a lower priority than the client threads. Communication delays have been set according to the maximum measured latencies in Fig. 6 for a message size of $1\,\mathrm{kB}$. See the lower part of Fig. 8 for all parameters. All other threads of the case study are allocated to other partitions. Other threads' execution does not affect the presented results thanks to the provided timing isolation between partitions.

Effect of worst-case service time. The worst-case service time of σ_2 is varied from 5 to $500\,\mathrm{ms}$ in steps of $5\,\mathrm{ms}$. Fig. 9a shows the resulting response time bounds under the setting `C-S` and `LOCAL-I` for both client threads. The Detection client experiences a constant improvement between the to cases, while the Localization client's improvement is growing with the worst-case service time of σ_2. This demonstrates the effectiveness of the inheritance mechanism for this setting.

Effect of distributed deployment. This experiment targets a comparison between the `C-S` and the `DISTR-I` setting. The Detection and Localization server are moved to a separate node and placed inside the system partition with a budget

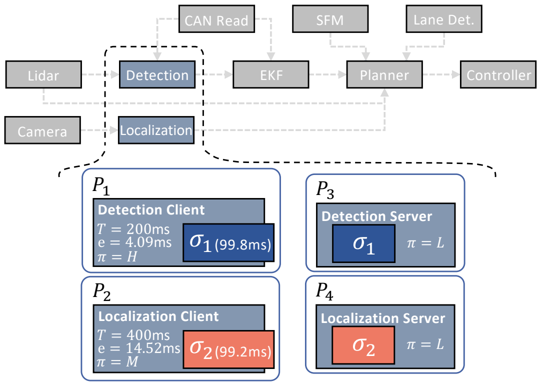
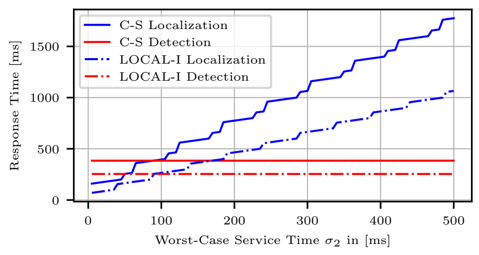
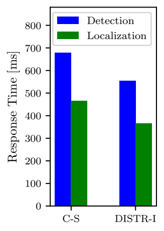


Figure 8: Adapted WATERS'19 case study with highlighted client/server allocation. Arrows among threads indicate data dependency but do not enforce precedence constraints.



(a)



(b)

Figure 9: (a) Varying WCST of σ_2 for `C-S` and `LOCAL-I`. (b) Comparison between `C-S` and `DISTR-I`.

of 100% (to comply with **A4**), together with an additional interfering thread τ_x with $T_x = 200\,\text{ms}$ and $e_x = 50\,\text{ms}$. The Detection server has been configured with the lowest priority, the Localization server with the second lowest priority, and τ_x has a higher priority than both servers. The priorities of both clients are higher than the one of τ_x.

The results are shown in Fig. 9b. In setting `C-S`, both servers receive interference from τ_x. In contrast, in the setting `DISTR-I`, servers inherit the priority of the client threads, which are higher than τ_x's priority. This allows for reducing the interference suffered by the server threads, leading to considerably smaller suspension-time bounds and hence WCRT bounds for the client threads.

VI. Related Work

To the best of our knowledge, only a few works studied the timing predictability of the QNX OS. The only two works are due to Dasari et al. [6, 8]. The first work [8] compared different configurations of the QNX OS in simulation. In the following paper [6], an end-to-end analysis for event chains under the APS scheduler of QNX has been proposed. However, none of these works considered the QNX SyncMP IPC and how it interacts with the APS scheduler.

Less close to our work, other research lines considered IPC protocols. For example, Brandenburg et al. [22] proposed an IPC protocol enabling temporal and logical isolation for mixed-criticality systems. Mergendahl et al. [23] studied the timing-related "Thundering Herd Attack" to synchronous IPC and budget management mechanisms, targeting the `seL4`-μ-kernel. Other works targeted inter-process communication by studying communication mechanisms and paradigms [24]–[28], while other proposals come from the context of hypervisors [29, 30] and separation kernels [31]–[34].

Priority-and-budget inheritance protocols have been studied for a long time, mainly in the context of accessing lock-protected shared resources [35]. The seminal work on priority inheritance is due to Sha et al. [36] in 1990. Later on, many protocols have been proposed to avoid priority-inversion effects, also considering applications protected by reservation servers. Most related to QNX are the approaches based on budget inheritance [37]–[40]. Many different other approaches have been proposed over the years [41]–[49] but all considering lock-protected shared resources rather than a client-server mechanism as in this paper.

VII. Conclusions and Future Work

This paper explored the SyncMP mechanism of QNX to implement the client-server paradigm. We performed an extensive set of experiments to derive a sound model for both local and distributed setups, considering the interaction of SyncMP with the APS reservation-based scheduler of QNX. We explored the behavior of the priority and partition inheritance features of APS, unveiling some un-documented behaviors with crucial implications on predictability. The model allowed us to derive a worst-case response time analysis for threads using the client-server paradigm. We also showed some key analysis challenges that arise on some of the considered setups, proposing guidelines (in the form of practically viable analysis assumptions) that system designers can conveniently use to overcome them and improve predictability. These challenges also highlighted that it would be beneficial for predictability if the QNX OS could provide options to enable priority and partition inheritance separately. The evaluation showed how, thanks to the analysis, it is possible to reason on the suitability of a system configuration (e.g., in terms of priority assignment and budget configuration), at design time, without the need to deploy the system on the actual platform using a profile-based empirical approach. This paper gives rise to a considerable number of research directions for future work. For example, more sophisticated analysis methods can be devised to deal with the challenges highlighted in Section IV-C and Section IV-D. Other research directions include the study of the real-time behavior of nested service requests and other IPC mechanisms of QNX, such as pulses and signals, extend the analysis to segmented or hybrid [50] self-suspending task models, and derive optimization algorithms to set the system parameters based on the analysis automatically.

Acknowledgment

We thank the reviewers for their insightful feedback that helped us improve the paper.

References

[1] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, 2018, pp. 287–296.

[2] A. Hamann, S. Saidi, D. Ginthoer, C. Wietfeld, and D. Ziegenbein, "Building end-to-end IoT applications with QoS guarantees," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[3] Z. Dong, W. Shi, G. Tong, and K. Yang, "Collaborative autonomous driving: Vision and challenges," in *2020 International Conference on Connected and Autonomous Driving (MetroCAD)*. IEEE, 2020, pp. 17–26.

[4] L. Belluardo, A. Stevanato, D. Casini, G. Cicero, A. Biondi, and G. Buttazzo, "A multi-domain software architecture for safe and secure autonomous driving," in *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2021, pp. 73–82.

[5] Blackberry QNX, *QNX® Neutrino® RTOS - System Architecture*, 2021.

[6] D. Dasari, M. Becker, D. Casini, and T. Blaß, "End-to-end analysis of event chains under the QNX adaptive partitioning scheduler," in *IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 214–227.

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

[8] 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.

[9] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley *et al.*, "Many suspensions, many problems: a review of self-suspending tasks in real-time systems," *Real-Time Systems*, vol. 55, no. 1, pp. 144–207, 2019.

[10] "Waters industrial challenge 2019," in *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2019.

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

[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] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, 2001, pp. 75–84.

[14] 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.

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

[16] G. Nelissen and A. Biondi, "The SRP resource sharing protocol for self-suspending tasks," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 361–372.

[17] Z. Dong, C. Liu, S. Bateni, K.-H. Chen, J.-J. Chen, G. von der Brüggen, and J. Shi, "Shared-resource-centric limited preemptive scheduling: A comprehensive study of suspension-based partitioning approaches," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2018, pp. 164–176.

[18] R. I. Davis, D. Griffin, and I. Bate, "A framework for multi-core schedulability analysis accounting for resource stress and sensitivity," *Real-Time Systems*, pp. 1–53, 2022.

[19] D. Casini, P. Pazzaglia, A. Biondi, and M. Di Natale, "Optimized partitioning and priority assignment of real-time applications on heterogeneous platforms with hardware acceleration," *Journal of Systems Architecture*, vol. 124, p. 102416, 2022.

[20] M. Boyer, P. Roux, H. Daigmorte, and D. Puechmaille, "A residual service curve of rate-latency server used by sporadic flows computable in quadratic time for network calculus," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[21] F. Rehm, D. Dasari, A. Hamann, M. Pressler, D. Ziegenbein, J. Seitter, I. Sañudo, N. Capodieci, P. Burgio, and M. Bertogna, "Performance modeling of heterogeneous HW platforms," *Microprocessors and Microsystems*, vol. 87, 2021.

[22] B. B. Brandenburg, "A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems," in *2014 IEEE Real-Time Systems Symposium*, 2014, pp. 196–206.

[23] S. Mergendahl, S. Jero, B. C. Ward, J. Furgala, G. Parmer, and R. Skowyra, "The thundering herd: Amplifying kernel interference to attack response times," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 95–107.

[24] P. Pazzaglia, D. Casini, A. Biondi, and M. D. Natale, "Optimizing inter-core communications under the LET paradigm using dma engines," *IEEE Transactions on Computers*, pp. 1–13, 2022.

[25] Y. Tang, X. Jiang, N. Guan, D. Ji, X. Luo, and W. Yi, "Comparing communication paradigms in cause-effect chains," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 82–96, 2023.

[26] M. Günzel, K.-H. Chen, N. Ueter, G. v. d. 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)*, 2021, pp. 40–52.

[27] A. Biondi and M. Di Natale, "Achieving predictable multicore execution of automotive applications using the LET paradigm," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 240–250.

[28] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication Centric Design in Complex Automotive Embedded Systems," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), pp. 10:1–10:20.

[29] G. Schwäricke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, "A real-time virtio-based framework for predictable inter-VM communication," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 27–40.

[30] N. Borgioli, M. Zini, D. Casini, G. Cicero, A. Biondi, and G. Buttazzo, "An I/O virtualization framework with I/O-related memory contention control for real-time systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2022.

[31] R. West, Y. Li, E. Missimer, and M. Danish, "A virtualized separation kernel for mixed-criticality systems," *ACM Trans. Comput. Syst.*, vol. 34, no. 3, jun 2016.

[32] Y. Li, R. West, Z. Cheng, and E. Missimer, "Predictable communication and migration in the Quest-V separation kernel," in *2014 IEEE Real-Time Systems Symposium*, 2014, pp. 272–283.

[33] M. Danish, Y. Li, and R. West, "Virtual-cpu scheduling in the quest operating system," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 169–179.

[34] E. Missimer, K. Missimer, and R. West, "Mixed-criticality scheduling with I/O," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2016, pp. 120–130.

[35] B. B. Brandenburg, *Multiprocessor Real-Time Locking Protocols*. Singapore: Springer Singapore, 2020, pp. 1–99.

[36] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[37] G. Lamastra, G. Lipari, and L. Abeni, "A bandwidth inheritance algorithm for real-time task synchronization in open systems," in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, 2001.

[38] D. de Niz, L. Abeni, S. Saewong, and R. Rajkumar, "Resource sharing in reservation-based systems," in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, 2001.

[39] D. Faggioli, G. Lipari, and T. Cucinotta, "The multiprocessor bandwidth inheritance protocol," in *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*, Brussels, Belgium, July 6-9, 2010.

[40] D. Faggioli, G. Lipari, and T. Cucinotta, "Analysis and implementation of the multiprocessor bandwidth inheritance protocol," *Real-Time Systems*, vol. 48, no. 6, pp. 789–825, 2012.

[41] T. M. Ghazalie and T. P. Baker, "Aperiodic servers in a deadline scheduling environment," *Real-Time Systems*, vol. 9, no. 1, pp. 31–67, 1995.

[42] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "SIRAP: a synchronization protocol for hierarchical resource sharing in real-time open systems," in *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT 2007)*, 2007.

[43] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS 2006)*, 2006.

[44] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Scheduling of semi-independent real-time components: Overrun methods and resource holding times," in *2008 IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2008)*, 2008.

[45] M. Behnam, T. Nolte, M. Sjodin, and I. Shin, "Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 1, pp. 93–104, Feb 2010.

[46] M. Bertogna, N. Fisher, and S. Baruah, "Resource-sharing servers for open environments," *IEEE Transactions on Industrial Informatics*, vol. 5, no. 3, pp. 202–219, 2009.

[47] 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.

[48] 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.

[49] Z. Tong, S. Ahmed, and J. H. Anderson, "Overrun-resilient multiprocessor real-time locking," in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, pp. 10:1–10:25.

[50] G. von der Brüggen, W.-H. Huang, and J.-J. Chen, "Hybrid self-suspension models in real-time embedded systems," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017, pp. 1–9.