# Compositional Virtual Timelines: Verifying Dynamic-Priority Partitions with Algorithmic Temporal Isolation

MENGQI LIU[*], Yale University, USA
ZHONG SHAO, Yale University, USA
HAO CHEN, Yale University, USA
MAN-KI YOON[*], Yale University, USA
JUNG-EUN KIM[*], Yale University, USA

Real-time systems power safety-critical applications that require strong isolation among each other. Such isolation needs to be enforced at two orthogonal levels. On the micro-architectural level, this mainly involves avoiding interference through micro-architectural states, such as cache lines. On the algorithmic level, this is usually achieved by adopting real-time partitions to reserve resources for each application. Implementations of such systems are often complex and require formal verification to guarantee proper isolation.

In this paper, we focus on algorithmic isolation, which is mainly related to scheduling-induced interferences. We address earliest-deadline-first (EDF) partitions to achieve compositionality and utilization, while imposing constraints on tasks' periods and enforcing budgets on these periodic partitions to ensure isolation between each other. The formal verification of such a real-time OS kernel is challenging due to the inherent complexity of the dynamic priority assignment on the partition level. We tackle this problem by adopting a dynamically constructed abstraction to lift the reasoning of a concrete scheduler into an abstract domain. Using this framework, we verify a real-time operating system kernel with budget-enforcing EDF partitions and prove that it indeed ensures isolation between partitions. All the proofs are mechanized in Coq.

CCS Concepts: • **Theory of computation → Program verification**; **Abstraction**; • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: earliest-deadline-first, dynamic-priority scheduling, partitioned scheduling, temporal isolation, formal verification, mechanized proof

**ACM Reference Format:**
Mengqi Liu, Zhong Shao, Hao Chen, Man-Ki Yoon, and Jung-Eun Kim. 2022. Compositional Virtual Timelines: Verifying Dynamic-Priority Partitions with Algorithmic Temporal Isolation. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 127 (October 2022), 29 pages. https://doi.org/10.1145/3563290

## 1 INTRODUCTION

Real-time systems often power safety-critical applications, such as avionics and automobile control systems, where stringent timing constraints must be satisfied. With the increasing trend of hosting multiple applications on common hardware, the underlying real-time operating system (OS) must

[*]Mengqi Liu is now at Alibaba. Man-Ki Yoon and Jung-Eun Kim are now at North Carolina State University

Authors' addresses: Mengqi Liu, mengqi.liu@yale.edu, Yale University, USA; Zhong Shao, zhong.shao@yale.edu, Yale University, USA; Hao Chen, hao.chen@yale.edu, Yale University, USA; Man-Ki Yoon, man-ki.yoon@yale.edu, Yale University, USA; Jung-Eun Kim, jung-eun.kim@yale.edu, Yale University, USA.

2475-1421/2022/10-ART127
https://doi.org/10.1145/3563290

guarantee not only the correct operation but also the proper scheduling of each individual component. This requires a systematic way of multiplexing both time and memory resources while preserving strict isolation between independent components.

## 1.1 Budget-Enforcing Partitions

Consider a self-driving car system comprised of multiple software components at different criticality levels. The GPS navigation module and the visual recognition module sit at the regular criticality level, carrying out normal driving operations. Under this layer, a collision-avoidance controller and a fail-safe controller constantly monitor the overall status of the driving and are ready to take over the vehicle once they detect immediate danger ahead. The latter two modules are at the highest-criticality level since they must function correctly even if the other parts of the system malfunction or are compromised. This requires strict isolation from the other components in the system. For example, these two modules can be placed in a secure execution environment so their memory states cannot be tampered with.

However, spatial isolation alone is not sufficient. The collision-avoidance controller must execute periodically to detect possible crashes in time. In other words, the controller must be reserved a fixed amount of execution time within each period. This is achieved by *hierarchical scheduling* [Aeronautical Radio, Inc. 2010; Davis and Burns 2005; Deng et al. 1997]. Here, the OS kernel encapsulates each application into a *time partition* to reserve its CPU share. Then it schedules partitions based on a certain policy (e.g., using priority), and an application can only execute when its enclosing partition is scheduled. Further, to prevent an overrun of one partition from jeopardizing the scheduling of the others, the OS kernel strictly enforces partition budgets at all times, thus ensuring that each application's budget requirement is satisfied regardless of the other partitions in the system. Sec. 2.1 explains more details about real-time hierarchical scheduling.

## 1.2 Verifying Temporal Isolation

Traditional real-time systems consist only of in-house components which are always trustworthy. Thus, they only need to focus on the time resource usage and guarantee that one component's ability to satisfy its timing requirement is not affected by behaviors of other components in the same system. However, a recent trend of integrating components from various vendors, some of which may be untrustworthy, which brings unique challenges regarding information flow security. This leads to a stronger notion of *temporal isolation*, which includes both real-time properties and information flow security. Sec. 2.2 provides more thorough discussion on these two different notions of temporal isolation.

Existing works mostly tackle spatial isolation and prove that memory resources are properly partitioned such that they do not lead to illegal information flow [Costanzo et al. 2016; Murray et al. 2013; Nelson et al. 2017].

*On the temporal aspect, existing works only address restricted settings and are not compositional.* For example, Murray et al. [2013] requires that partitions be static and scheduled in a round-robin manner. Liu et al. [2020] allows non-static partitions scheduled by a fixed-priority scheduling, but does not allow arbitrary partition periods. All these constraints limit their ability to compose diverse real-time applications into one system.

Let us consider the following example that highlights the *non-compositionality* of the fixed-priority (FP) scheduling. Under FP, a set of $n$ partitions is guaranteed to be schedulable if their total utilization ($\Sigma \frac{budget}{period}$) is not greater than $n(2^{1/n} - 1)$ (called the Liu-Layland bound) [Liu and Layland 1973]. Assume that initially, there are two partitions, $\Pi_0$ and $\Pi_1$, whose periods are 30 ms and 40 ms, respectively, and budgets are 10 ms, which leads to the total utilization of 58.33%. Because it is

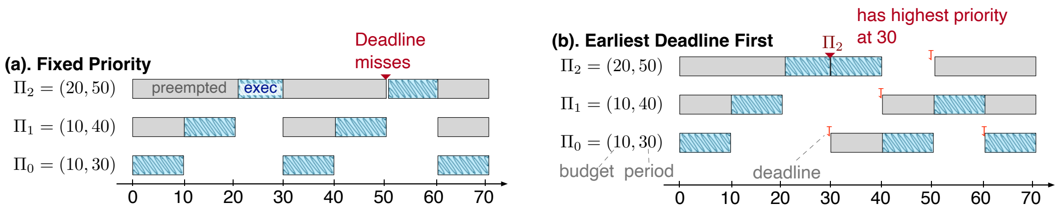


Fig. 1. Compositionality of Scheduling

less than the above bound (82.8% for $n = 2$), the partitions are schedulable under FP. Now, adding another partition whose period and budget are 50 ms and 20 ms, respectively, will increase the total utilization to 98.33%, while the Liu-Layland bound changes (77.97% for $n = 3$).[1] Fig. 1(a) shows that $\Pi_2$ misses its deadline at time 50ms.

The above threshold depend on the number of partitions in the system. Further, it relies on the assumption that partitions with larger periods occupy lower priorities. Otherwise, an exact analysis [Lehoczky et al. 1989] will need to take into account the priority, period and budget of all partitions. This is non-compositional, where a compositional scheme will only need to consider the total utilization of partitions alone, ignoring their numbers and exact parameters.

This non-compositionality is an inherent nature of the FP algorithm, which favors higher-priority partitions over lower-priority ones at any instant. In fact, partitions of different priorities can be viewed as layered on top of each other, where lower-priority ones can only execute when the higher-priorities ones have finished. From the verification perspective, this recursive structure is leverage for the formal reasoning of an FP scheduler, since it maps naturally to an inductive proof. Sec. 2.3 explains in more detail how this recursive structure is characterized by *virtual timeline* [Liu et al. 2020] and is used in the formal verification of an FP scheduler.

For performance, such non-compositionality may easily lead to *underutilization* of the processor, i.e., partitions cannot make 100% reservation of the CPU resource, especially with the trend of hosting multiple applications with arbitrary periods on a common hardware. Real-time applications often sample data periodically from devices (e.g., sensor input). In this case, a device's updating rate becomes the base period for its enclosing partition, whose other tasks follow periods that are multiples of this base rate. A complex real-time system may consist of multiple components with diverse base rates and hence may suffer a waste of CPU time due to this non-compositionality.

*Earliest-deadline-first (EDF) scheduling is compositional, but significantly more challenging to verify.* EDF is a dynamic-priority assignment algorithm, which eliminates fragmentation of CPU time and thus allows the free composition of components (even with diverse periods) as long as their total utilization does not exceed 100% [Liu and Layland 1973]. As illustrated in Fig. 1(b), under EDF, partition $\Pi_2$ shifts to the highest priority at time instant 30 and is thus able to schedule for its full budget. Further, we adopt the budget enforcement mechanism to prevent the overrun and malfunctioning within one partition from affecting the scheduling of other partitions.

However, from the verification perspective, the same mechanism brings significant challenges to the formal reasoning of OS kernels that adopt an EDF scheduler, e.g. for partition-level scheduling. Such verification is more challenging than a standalone schedulability analysis, because real-time properties need to connect with the concrete scheduler implementation and serve as pre-conditions for other OS functionalities. Because priority between components varies at different time instants

[1]It is a sufficient condition. If the utilization bound test fails, the partitions need to be tested against the exact analysis [Joseph and Pandya 1986].

and partitions can no longer be viewed as layering on top of each other, the recursive structure found in FP no longer holds under EDF. Thus, it is beyond the capability of static virtual timelines [Liu et al. 2020].

### 1.3 Contributions

This paper makes the following contributions:

- We present a novel language-based abstraction to reason about the concrete implementation of preemptive schedulers with a *dynamic priority assignment*. In particular, it adopts a dynamic computation to address *braided timelines* which are free to intersect or shift priorities and are not limited to a fixed hierarchy. It achieves compositionality in the sense that real-time components with diverse periods are free to reside in the same system as long as their total utilization does not exceed 100%.
- Using this abstraction, we implement and verify a real-time OS kernel with budget-enforcing EDF partitions. We prove that partitions can make a full reservation of the CPU resource. We prove *temporal isolation* of partitions (conditioned on the task arrival model) by showing that the task schedule inside a partition is oblivious of others. All our proofs are mechanized in Coq and carried down to the generated assembly code.

## 2 BACKGROUND AND CHALLENGES

In this section, we provide background on real-time hierarchical scheduling, temporal interference between partitions, and the static *virtual timeline* for verifying FP schedulers.

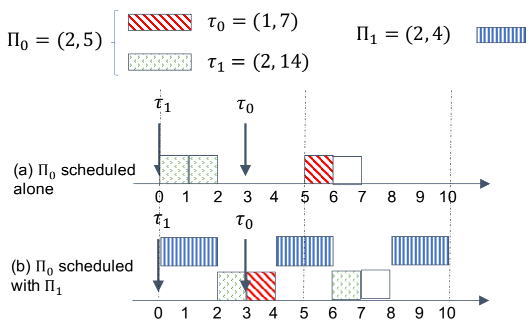
### 2.1 Background: Real-Time Partitions and Hierarchical Scheduling

In real-time systems, a *task* is the basic unit of execution, corresponding to a user-level process hosted by an OS kernel. It is associated with a pre-specified *period* and *budget* (a.k.a. worst-case execution time), making its schedule different from a non-real-time process. For example, a task with a period of 500 ms and a budget of 60 ms will run for 60 ms within every period of 500 ms, say, from time 0 to 500 ms, from 500 ms to 1000 ms, etc.

Multiple tasks constitute a real-time *application* so that they cooperate to achieve specific functionality. However, to accommodate the diverse periods and budgets of tasks and isolate one application's time resource usage from another, an application is usually associated with a time *partition*. A partition is also associated with a pre-specified period and budget so that the accumulative execution time of its tasks is bounded by the partition budget within every consecutive time window of the partition period. Fig. 3 summarizes the notations used in this paper.

Fig. 2(a) illustrates how a partition, $\Pi_0$, and its two tasks, $\tau_0$ and $\tau_1$, are scheduled. Task $\tau_0$ has a period of 7 and a budget of 1, such that it will execute for 1 time slot within the time window [0, 7]. Similarly, $\tau_1$ executes for 2 time slots within [0, 14]. Their enclosing partition, $\Pi_0$, guarantees that the collective execution time of $\tau_0$ and $\tau_1$ does not exceed 2 time slots within [0, 5], [5, 10], etc. Notice that, in the second partition period, since both $\tau_0$ and $\tau_1$ have used up their budgets and thus cannot execute, an idle task is scheduled to occupy the time slot allocated for $\Pi_0$, represented by the white box.

In real-time hierarchical scheduling, an application partition is first selected by the global scheduler, and then a task inside the partition is selected by its local scheduler. There are two approaches to real-time hierarchical scheduling. Static partitionings, which can be found in avionics systems [LynxOS-178 2022; VxWorks 653 2022], follow a fixed partition schedule repetitively. They are simple to implement and guarantee strong isolation between components. However, it is a fundamental limitation that this scheme is inflexible in CPU usage [Heiser 2020; Kim et al.



(2, 5) means the component's budget is 2 and period is 5. Downarrow represents the arrival of a task.

Fig. 2. Temporal interference among EDF partitions

| Symbol | Description |
|---|---|
| $N$ | The number of partitions |
| $\Pi_i$ | The $i$-th partition, where $0 \leq i < N$ |
| $T_i$ | The period of $\Pi_i$ |
| $C_i$ | The time budget of $\Pi_i$ within each period |
| $\pi_i$ | The virtual time map for $\Pi_i$. It is of type $Z \rightarrow Z$ |
| M | The number of tasks inside a particular partition |
| $\tau_j$ | The $j$-th task inside a particular partition, where $0 \leq j < M$ |
| $p_j$ | The period of $\tau_j$ |
| $e_j$ | The budget of $\tau_j$ within each period |

Fig. 3. Notations

2015]. Further, supporting applications with different base rates can be challenging because static-partitioning relies on finding a major cycle that accommodates the cyclic behavior of all components.

On the contrary, non-static partitioning allows enhanced resource flexibility and results in better responsiveness. This category employs real-time server algorithms (such as periodic server [Shin and Lee 2003], sporadic server [Sprunt et al. 1989], constant bandwidth server [Abeni and Buttazzo 1998], which is adopted by the SCHED_DEADLINE mechanism in Linux) to contain a set of tasks. A server (used interchangeably with a partition) is characterized by the budget, $C_i$, and the replenishment period, $T_i$. The server's budget is consumed as much as its tasks are executed. Partitions can be scheduled based on fixed-priority (e.g., periodic server, sporadic server) or dynamic-priority (e.g., constant bandwidth server).

## 2.2 Challenge: Temporal Interference Between Non-Static Partitions

As explained above, time partitioning reserves a time budget for a group of tasks, so that they can meet their own timing requirements (e.g., guaranteed amount of time budget within each period) no matter how tasks in other partitions behave. This notion of *temporal isolation* is widely adopted in the real-time community. Indeed, real-time systems usually host multiple components with diverse timing requirements. Since all components are trusted, traditional spatial isolation techniques (e.g., virtual memory) may not be as important for the system. Instead, such a system usually focuses on temporal isolation and adopt static or dynamic time partitioning schemes to distribute time resource properly among components. As mentioned above, non-static partitioning achieves better CPU utilization and is extensively studied in the literature.

However, in recent years, there is a growing trend of integrating various components, not necessarily from the same vendor, onto a single system [Aeronautical Radio, Inc. 2010; Yoon et al. 2021]. This achieves better cost-effectiveness but also exposes benign components to potential confidentiality violations: malicious components can collude to infer secret information from them since they share common CPU resources. Although various spatial isolation techniques exist, they cannot handle challenges caused by real-time scheduling schemes (e.g., temporal interference caused by the non-static time partitioning). This is a unique challenge for real-time systems which also require isolation in the sense of *security*.

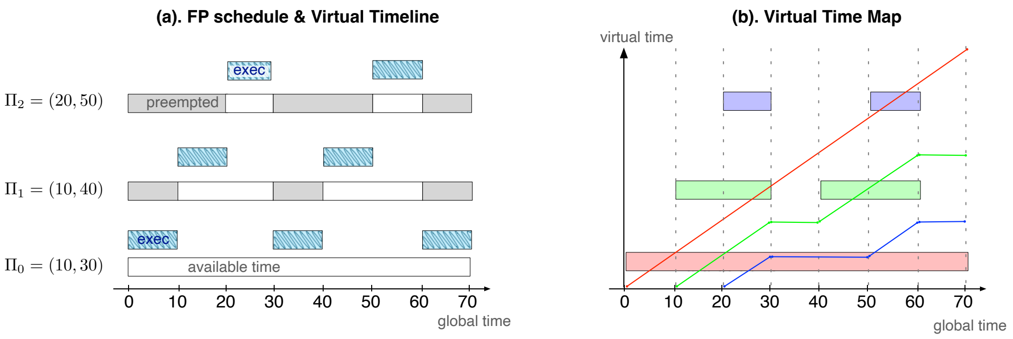


Fig. 4. Static virtual timeline

Let us consider Fig. 2 again. Here, partitions are scheduled by EDF. We examine partition $\Pi_0$, which consists of two tasks, $\tau_0$ and $\tau_1$, where $\tau_0$ has a higher priority. Case (a) depicts the case when $\Pi_0$ runs alone. In this case, it always occupies the first 2 slots within each period (= 5). As a comparison, case (b) schedules $\Pi_0$ together with $\Pi_1$. Though $\Pi_0$ is still guaranteed 2 time slots in every period, its exact schedule differs from (a). Specifically, the relative ordering between $\tau_0$ and $\tau_1$ is flipped depending on the exact schedule of their enclosing partition $\Pi_0$. This is caused by the varying task arrival time in this partition's local view, which is affected in turn by the behavior of the other partition, i.e., $\Pi_1$. Such non-determinism may allow a partition (e.g., $\Pi_0$) to infer the execution pattern of another (colluding) partition (e.g., $\Pi_1$) by observing differences in its own task schedule, which effectively creates a covert channel, breaking the information-flow policy. Liu et al. [2020] formalized this problem, and Yoon et al. [2021] demonstrated the feasibility of establishing such covert channels (e.g., leaking out sensitive data item) in a Linux-based real-time operating system that operates on a 1/10th-scale self-driving car platform.

This paper focuses on the above notion of *temporal isolation* in the sense of *information flow security*. Notice that this is a unique challenge for real-time systems that need to integrate potentially untrusted components and execute them alongside trusted ones. For all other real-time systems, the traditional notion of real-time temporal isolation suffices to guarantee their correct operations – they do not suffer the same security vulnerability if all components are trusted.

## 2.3 Background: Static Virtual Timeline and Real-Time CertiKOS

CertiKOS [Costanzo et al. 2016; Gu et al. 2015, 2019, 2016] is a verified OS kernel featuring a formal functional correctness guarantee down to the assembly code level. It centers around the idea of abstraction layers, which formally connects a piece of C/assembly code with its Coq specification through a simulation proof. Using this layered approach, CertiKOS builds a verified OS kernel and proves that the behavior of its concrete implementation is equivalent to its Coq specification.

The real-time extension of CertiKOS [Liu et al. 2020] proposes the *static virtual timeline* to describe and reason about the temporal behavior of a fixed-priority (FP) scheduler. Their main observation is that there is a fixed precedence relation among components, such that the preemption of a component can be viewed as hiding certain time slots away from it. For example, Fig. 4(a) illustrates the FP scheduling of three partitions, $\Pi_0$, $\Pi_1$, and $\Pi_2$, in decreasing order of priority. The bottom plot depicts each partition's virtual timeline, i.e., which time slots are available to it (white boxes) and which ones are not (grey boxes). The top plot depicts the execution of the partition, which must correspond to white boxes on its virtual timeline. The virtual timeline for $\Pi_0$ contains only white boxes because it has the highest priority and can never be preempted. Partition $\Pi_1$

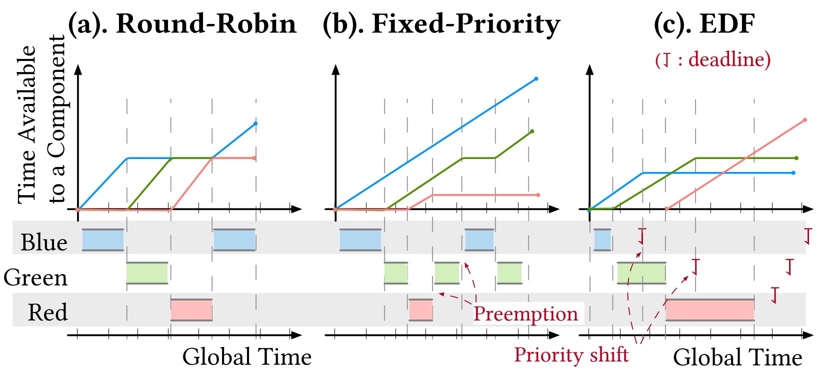


Fig. 5. Temporal behavior of independent partitions under different scheduling policies. (Each colored line represents an independent partition.)

is only preempted by $\Pi_0$, thus grey boxes on its virtual timeline correspond to the execution of $\Pi_0$. The remaining time slots are available to $\Pi_1$ and are depicted as white boxes. Finally, $\Pi_2$ is preempted by both $\Pi_0$ and $\Pi_1$, so that grey boxes on its virtual timeline correspond to both tasks' execution. In this way, the virtual timeline hides interferences from higher-priority partitions.

To facilitate mechanized formal reasoning of virtual timelines, Liu et al. [2020] use a virtual time map to describe a virtual timeline. Such a concept is similar to the supply-bound functions in the real-time literature [Lehoczky et al. 1989; Liu 2000], but with adapted definitions to facilitate verification of the scheduler's source code. Here, virtual time is the number of available time slots for a partition, i.e., the total length of white boxes, up to a particular global time instant. Virtual time map is a mapping from the global time to the virtual time of a partition, denoted as $\pi_i$. For example, all time slots are available to the highest-priority partition, $\Pi_0$. Thus,

$$\pi_0(t) = t$$

Time slots not occupied by $\Pi_0$ are available to the second-highest one, $\Pi_1$, and so on. Assume that a partition $\Pi_i$ becomes ready to run at regular time instants $0, T_i, 2T_i, 3T_i, \ldots$, and that it executes for $C_i$ time slots within each period. The virtual time map for $\Pi_{i+1}$ can be constructed as follows (adapted from Def. 2.1 in [Liu et al. 2020]).

*Definition 2.1.* The static construction of time maps for fixed-priority scheduling:

$$\pi_{i+1}(t) = \pi_i(t) - \lfloor \frac{t}{T_i} \rfloor C_i - min(C_i, \pi_i(t) - \pi_i(\lfloor \frac{t}{T_i} \rfloor T_i))$$

This means $\pi_{i+1}(t)$ comes from $\pi_i(t)$, excludes the execution of $\Pi_i$ in previous whole periods $(\lfloor \frac{t}{T_i} \rfloor C_i)$ joints with the execution of $\Pi_i$ in the last partial period. This way of static construction of time maps is specialized in the *fixed*-priority setting. Liu et al. [2020] prove that such an abstraction faithfully captures the temporal behavior of components. Thus the reasoning of a concrete fixed-priority scheduler implementation is lifted to an abstract domain, where they formally prove temporal properties such as schedulability and obliviousness to lower-priority components.

## 2.4 Challenge: Non-Compositionality of Static Virtual Timeline

Formally verifying an EDF scheduler on the source code level is challenging. Though virtual timeline lifts the verification of a concrete scheduler into the abstract domain, getting it to reason about the EDF scheduling is highly non-trivial. The bottom plots in Fig. 5 give an empirical illustration of individual partition's temporal behavior under different scheduling algorithms. Here, the horizontal axis represents the global time, and the vertical axis represents the time available to a particular partition. The overall map is denoted as a virtual timeline. In Round-Robin scheduling, a time slot is only available to one partition at any instant, and thus time resources for different partitions do not overlap. In FP scheduling, time resources are allocated to partitions in decreasing order of fixed-priority, resulting in properly layered timelines in the figure (refer to Def. 2.1 for the exact computation). This recursive nature enables the inductive proof of the FP scheduler in [Liu et al. 2020].

On the other hand, in EDF scheduling, the priority between partitions varies at different time instants, as shown in the top plot (Fig. 5(c)). In this way, the resulting timelines become *braided*, i.e., they overlap and intersect with each other instead of being properly layered. Such lacking of a recursive structure makes its formal reasoning significantly more challenging, which is beyond the capability of static virtual timelines [Liu et al. 2020].

We tackle this problem by adopting the dynamic computation of time maps. This allows us to separate the source code of an EDF scheduler from the high-level reasoning of its temporal behavior. The intuition is that, even though there is no fixed hierarchy among partitions, at any particular moment, a transient priority relation still exists and is deterministic. We make it explicit by calculating a priority queue at each invocation of the scheduler. This allows us to dictate the availability of a time slot. For example, if partition $\Pi_i$ is scheduled on time slot $[t, t+1)$, obviously this slot is available to $\Pi_i$. Further, it is also available to partitions with higher priorities. In fact, the reason this slot is allocated to $\Pi_i$ is that all higher-transient-priority partitions have used up their budgets, and thus this slot is available to but not used by them. We then embed the dynamic computation of timelines in a virtual-time-based scheduler. At any moment $t$, the scheduler iterates over all partitions in the transient priority queue and schedules the first one available to execute on its virtual timeline. Finally, the scheduler updates time maps for all partitions up to the next time tick, $t + 1$. This is illustrated in Fig. 9. We explain in more detail in Sec. 4.

## 3 OVERVIEW OF OUR APPROACHES

Previous sections explain two major problems with existing verified real-time OS kernels when being used to integrate real-time components from multiple vendors: (1) the application of hierarchical scheduling may lead to inter-partition interferences (Sec. 2.2); (2) existing formal technique, i.e., static virtual timeline, is not compositional and only works with straightforward scheduling algorithms, such as FP (Sec. 2.4). In this section, we explain our overall design of a secure OS kernel with budget-enforcing EDF partitions, and address the challenges in its formal verification.

### 3.1 Threat Model

In this work, we address an *algorithmic covert channel* in the context of hierarchical scheduling in a real-time OS kernel. The attacker controls one or more partitions in the system and tasks inside these partitions. The attacker's tasks can access its own memory, invoke system calls (e.g., to communicate with each other inside the same partition), and thus observe the order in which its tasks are scheduled. Indeed, this allows an attacker to measure the progress of its own tasks and to indirectly infer the passage of global time. However, the attacker's code cannot read the global time or access time-leaking devices directly. Thus, the attacker's measurement of time is still logical. Communication between two partitions is also prohibited.

Notice that a real-time task's proper functioning does not necessarily rely on having direct access to a real clock. The OS is able to schedule the task according to its timing parameters. Further, there are various mechanisms for preventing access to a real clock. On the instruction level, modern processors provide mechanisms to prevent user-level instructions to access timing registers, e.g. through RDTSC. On the system level, the OS can properly restrict access to time-leaking devices.

The attacker observes variations in its own task schedule to infer the behavior of tasks in other partitions [Yoon et al. 2021]. Such a covert channel is *algorithmic* because it leverages the scheduling algorithm instead of *micro-architectural-level* details. To close such a channel, this work aims to prove the confidentiality property of the scheduler implementation, i.e., the attacker cannot infer information about other partitions by observing its own scheduling behavior.

## 3.2 Implementation Approaches

We make the following design decisions to achieve temporal isolation and full reservation of CPU resources.

- Firstly, budget enforcement prevents the overrun of one partition from affecting other partitions' CPU usage, which is necessary for ensuring isolation.
- Secondly, since applications may rely on devices with arbitrary base rates, our system must accommodate arbitrary partition periods. In this case, FP suffers fragmentation issues and thus cannot make full reservation of the CPU resource in a general scenario, as explained earlier. On the contrary, EDF achieves 100% reservation of the CPU resource while allowing arbitrary partition periods. Thus, we adopt EDF for the partition-level scheduling.
- Lastly, we require tasks to be *bound* by their enclosing partitions to avoid covert channel between partitions as illustrated in Fig. 2. More specifically, we require that tasks' periods are multiples of the partitions' period and that tasks' arrivals are always synchronized with the boundaries of the partition's periods [Davis and Burns 2005]. This is in fact a common practice in real-time systems. Sec. 6.2 provides more lengthy discussions.

## 3.3 Verification Approaches

Even though the above implementation approaches avoid the covert channel illustrated in Fig. 2, its formal verification remains challenging. Fig. 6 gives an overview of our approach. Here, each rectangle represents one "layer," which is a collection of functions and internal states of the OS kernel. Yellow boxes represent concrete C functions. Blue boxes represent abstract functions in Coq. The green arrow, $\sqsubseteq$, denotes contextual refinement proofs between two layers, which abstracts C functions into Coq representations, or abstracts Coq representations into forms that are more suitable for reasoning.

As shown in Fig. 6, the partition-level scheduler `psched` and local scheduler `lsched` are both available in the first layer, "Concrete Scheduler". However, proving schedulability and temporal isolation directly in this layer will be too challenging. Instead, we introduce new intermediate layers to bridge the gap between "Concrete Scheduler" and "Abstract Scheduler with Oblivious lsched", so that proofs for each step can be done more easily. In the following, we explain the intuition behind each layer.

*Layer "Concrete Scheduler".* This layer includes two pieces of C code, the partition-level scheduler, `psched` (Fig. 7), and the local scheduler, `lsched` (Fig. 17). They operate on the global time `t`, the partition-level quanta array $Q_{partition}$, and the task-level quanta array $Q_{task}$.

In particular, `psched` first refills budgets for partitions that are about to start a new period (Line 4 - 8). It then iterates over all partitions in the order of their index (Line 13), and maintains *pid* and *ddl* to find the partition with positive remaining budget and the nearest deadline (Line 14 - 21). Here, partition $\Pi_i$'s deadline is the ending time of its current period, as computed on Line 15. Finally, `psched` deducts $Q_{partition}[pid]$ if $\Pi_{pid}$ is not an idle partition (Line 23 - 25).

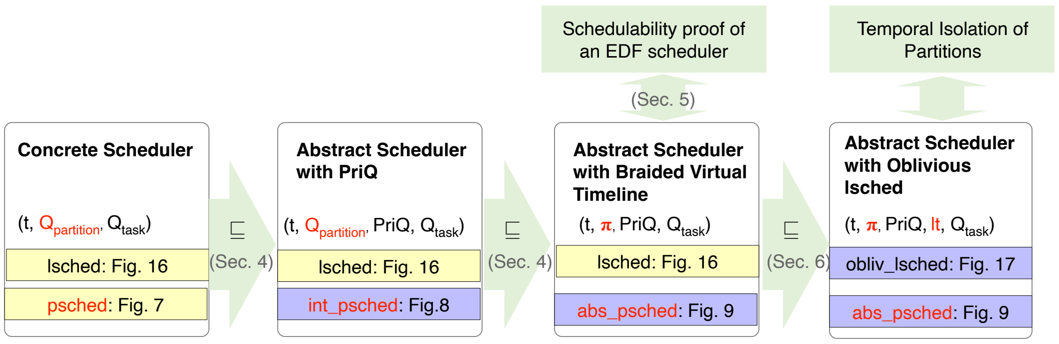


Fig. 6. Overview of our verification

```
int psched(){
  t++;
  // refill partition budgets
  for(int i = 0; i < N; i++){
    if (t % T_i == 0){
      Q_partition[i] = C_i;
    }
  }

  // scheduling
  int pid = N;
  int min_ddl = INT_MAX;
  for(int i = 0; i < N; i++){
    if (Q_partition[i] > 0){
      int ddl = t / T_i * T_i + T_i;
      if (pid == N ||
          ddl < min_ddl){
        pid = i;
        min_ddl = ddl;
      }
    }
  }
  if (pid < N){
    Q_partition[pid]--;
  }
  return pid;
}
```

Fig. 7. C code of a concrete EDF scheduler

```
1  int int_psched(){
2    t++;
3    // refill partition budgets
4    for(int i = 0; i < N; i++){
5      if (t % T_i == 0){
6        Q_partition[i] = C_i;
7      }
8    }
9
10   // scheduling
11   int pid = N;
12   for(int p = 0; p < N; p++){
13     int i = PriQ_t(p);
14     if (Q_partition[i] > 0){
15       pid = i;
16       break;
17     }
18   }
19   if (pid < N){
20     Q_partition[pid]--;
21   }
22   return pid;
23 }
```

Fig. 8. C demonstration of the intermediate scheduler (actually specified in Coq)

Even though the implementation of the above EDF scheduler is straightforward, its formal reasoning is extremely challenging, as explained in Sec. 2.4. We introduce two more layers as shown below to facilitate its formal proof.

*Layer "Abstract Scheduler with PriQ".* It introduces a ghost state, *PriQ* (see Fig. 10), to represent the transient priority queue at any time instant (the closest deadline corresponds to the highest priority). As shown on Line 12 in Fig. 8, $p$ represents the transient priority level, where 0 means the highest priority. On Line 13, $PriQ_t$(p) returns the partition index i at priority level p. Unlike psched, `int_psched` (Fig. 8) iterates over all partitions in decreasing order of priority and checks whether the remaining budget is positive or not. If $\Pi_i$ is found at Line 15, it must have the closest

```
int abs_psched(){
  t++;

  int pid = N;
  for (p = 0; p < N; p++) {
    int i = $PriQ_t$(p);
    if (pid == N){
      $\pi_i = \lambda x.(x \geq t) \ ? \ \pi_i(t-1)+1 : \pi_i(x);$
      if ($(\pi_i(t) - \pi_i(\lfloor t/T_i \rfloor T_i) < C_i$){
        pid = i;
      }
    }else{
      // Update $I_i$ instead of $\pi_i$
      $I_i(\lfloor t/T_i \rfloor)$++;
    }
  }
  return pid;
}
```

Fig. 9. C illustration of the dynamic construction of virtual time maps (actually specified in Coq)

deadline among partitions that are ready to run. Thus, `int_psched` indeed satisfies the EDF policy. We prove that `int_psched` is contextually equivalent [Gu et al. 2015] to `psched` in Sec. 4.

*Layer "Abstract Scheduler with Braided Virtual Timeline".* Here, we define compositional virtual timelines ($\pi$, explained in more detail in Sec. 4) to characterize the temporal behavior of EDF scheduling. $I_i(\lfloor t/T_i \rfloor)$ is another ghost state that tracks the amount of interference in the $\lfloor t/T_i \rfloor$-th period to facilitate the formal reasoning on $\pi$, which is explained in more detail in Sec. 5. We then prove that the intermediate EDF scheduler, `int_psched` (relying on array $Q_{partition}$), is contextually equivalent to `abs_psched` (Fig. 9), such that $\pi$ is indeed a faithful abstraction (Sec. 4.2).

This allows us to address the formal reasoning of the concrete EDF scheduler by formalizing its behavior on top of $\pi$. Sec. 5.1 explains in more detail how such a notion facilitates the proof that each partition gets its full budget within every period. In this way, through equivalence proofs across three consecutive layers (`psched` $\sqsubseteq$ `int_psched` $\sqsubseteq$ `abs_psched`), the property specified using an abstract $\pi$ carries down to the concrete source code of the EDF scheduler.

*Layer "Abstract Scheduler with Oblivious lsched".* Finally, this layer addresses the task-level scheduler inside partitions, `lsched` (Fig. 17). We prove its contextual equivalence to a Coq abstraction, `obliv_lsched` (Fig. 18), which maintains the partition's local time and is oblivious to the global time (Sec. 6.3). In this way, we prove that the local behavior of partitions is independent of other partitions residing on the same processor, and formally close the algorithmic covert channel illustrated in Sec. 2.2.

## 4 BRAIDED VIRTUAL TIMELINE

In this section, we explain how we use braided virtual timelines to reason about EDF scheduling on the partition level.

### 4.1 Construction of the Braided Virtual Timeline

Fig. 9 illustrates the dynamic construction of braided virtual timelines. `N` denotes the number of partitions in the system. Here, $\pi_i$(t) denotes the amount of time available to partition $\Pi_i$ in the global time range $[0, t)$. However, unlike the static construction in Def. 2.1, we cannot compute in advance the value of $\pi_i$ at every time instant. Rather, at time t, all time maps ( $\{\pi_0, \pi_1, ..., \pi_{N-1}\}$ ) are

```
 1  Inductive priEntry: Type :=
 2  | PriEntry: Z (* deadline *) → Z (* partition ID *) → priEntry.
 3
 4  Function priEntry_le e1 e2 :=
 5    match e1, e2 with
 6    | PriEntry d1 id1, PriEntry d2 id2 ⇒
 7      if (zlt d1 d2) then true
 8      else (if zeq d1 d2 then
 9              (if zle id1 id2 then true else false) else false)
10    end.
11
12  (* A list of entries, from high to low priority  *)
13  Definition priQueue := list priEntry.
14
15  (* Partition parameters *)
16  Inductive par_config: Type :=
17    | ParValid: Z (* period *) → Z (* budget *) → par_config.
18
19  Fixpoint get_priqueue_aux (N: nat) (t: Z) (conf: ZMap.t par_config):=
20    match N with
21    | O ⇒ nil
22    | S n ⇒
23      match (ZMap.get (Z.of_nat n) conf) with
24      | ParValid T C ⇒
25        PriEntry (t / T * T + T) (Z.of_nat n) :: (get_priqueue_aux n t conf)
26      end
27    end.
28
29  Definition get_priqueue (N: nat) (t: Z) (conf: ZMap.t par_config) :=
30    sort priEntry_le (get_priqueue_aux N t conf).
```

Fig. 10. Coq formalization of the priority queue

valid up to t. Then the scheduler increments the time counter to t + 1 and iterates over all partitions in decreasing order of transient priority (p from 0 to N-1). Variable pid denotes the partition to be scheduled. Initially, it equals N (Line 4), meaning the idle partition will be selected by default. As the scheduler iterates over partition $\Pi_i$ (Line 6 - 12), it increments $\pi_i$ by 1 at time t+1 (Line 8) because the time slot [t, t+1) is available to $\Pi_i$. If $\Pi_i$ is not finished, i.e., the amount of time available to it at the current period is less than its budget (Line 9), it will be selected by the scheduler because it has the highest transient priority among ready partitions. Then, for lower-priority partitions, their virtual time stagnates because they are preempted (Line 13). Here, on Line 14, $I_i[t/T_i]$ is a ghost state that records the amount of interference $\Pi_i$ experiences in each of its period. We will explain in more detail in Lemma 5.5. In this way, if $\Pi_{pid}$ is scheduled, the virtual time for itself and higher-priority partitions is incremented, while the virtual time for lower-priority ones stagnates.

The intuition behind the loop in Fig. 9 is that at each moment t, there is still transient priority among partitions, allowing us to define the preemption relation between them. In particular, at any moment `t`, we compute a priority queue $PriQ_t$, which orders all partitions by their current deadlines. Fig. 10 shows its Coq formalization. Here, a partition is represented as `ParValid T C`, where T is its period and C is its budget. At moment t, `get_priqueue_aux` computes a list of `PriEntry d id`, where id is partition ID and d denotes the current deadline of $\Pi_{id}$ (Line 25). The remaining work is to sort this list by each partition's priority, i.e., deadline. This is achieved by invoking sort with a custom comparator (Line 30), `priEntry_le`, which first compares the two deadlines, and then compares the partition IDs as a tie-breaker. In the end, `get_priqueue` returns the priority queue for the set of partitions `conf` at time instant `t`. We observe that the transient priority relation is indeed pre-defined and does not depend on the runtime behavior, e.g. whether a task has used up its budget or not at any particular moment.

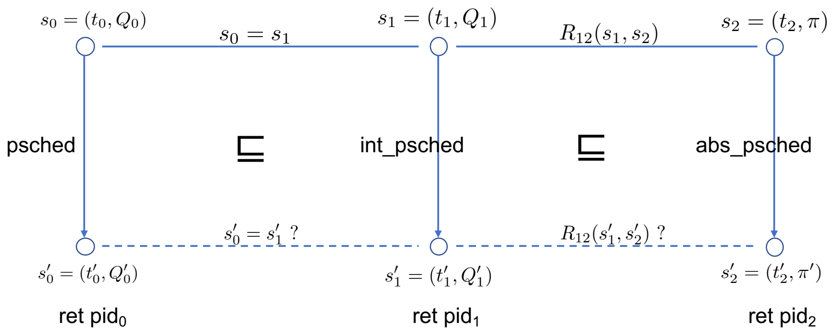


Fig. 11. Contextual refinement proof

This leads to the same motivation when the virtual timeline is first introduced for fixed-priority scheduling: encapsulate interferences from higher-priority partitions such that the schedule of one partition can be decided by looking at its virtual timeline alone. In particular, Line 9 in Fig. 9 compares $C_i$ with $\pi_i(t) - \pi_i(\lfloor t/T_i \rfloor T_i)$. If the former is smaller, partition $\Pi_i$ has been given enough available time slots during the current period and must have used up its budget. Otherwise, $\Pi_i$ is ready to run at the next time slot.

This formalization helps us decouple the scheduler's source code from its high-level reasoning. Fig. 7 demonstrates the source code for an EDF scheduler. It uses an array, $Q_{partition}$, to track the remaining budget of each partition. At any moment, it iterates over all partitions to find the one with a positive budget and the nearest deadline. The formal connection between virtual timelines and this scheduler implementation relies on a refinement proof that the two are equivalent in deciding a task's schedule. The proof structure is similar to that in [Liu et al. 2020], and we explain more details in Sec. 4.2.

## 4.2 Contextual Refinement with the Concrete Scheduler

In this section, we explain how the braided virtual timeline is formally connected to the source code of an EDF scheduler. This step corresponds to the refinement proof between the partition-level EDF scheduler (psched in Fig. 7) and the braided virtual timeline abstraction (abs_psched in Fig. 9) shown in Fig. 6.

In this paper, we follow the certified abstraction layer approach [Gu et al. 2015] to verify the source code of a real-time OS kernel. Here, a refinement proof between a lower-layer function and a higher-layer function requires matching the relevant state and return value of both functions. In particular, starting from equivalent states, the two functions must execute and result in equivalent final states and yield identical return values.

Fig. 11 illustrates the refinement proof between psched and abs_psched. Here, psched operates on the global time t and the quanta array $Q$, while abs_psched operates on t and the virtual time maps $\pi$. After executing both functions, they will update relevant states and return the pid for the next partition to schedule. Since the two functions rely on different definitions of states, we use $R_{12}$ to denote their equivalence relation, which is more sophisticated than being identical. Thus, the goal of the refinement proof is to show that if the two starting states satisfy $R_{12}(s_0, s_2)$, then the two final states must also satisfy such equivalence property, $R_{12}(s'_0, s'_2)$. Notice that an OS kernel maintains many more states than the global time and quanta array for its scheduler. In the actual proof, all other states are identical across $s_0$ and $s_2$ and remain unchanged in $s'_0$ and $s'_2$.

| | psched | int_psched | abs_psched |
|---|---|---|---|
| How the scheduler iterates over all partitions | by partition ID | by the transient priority | by the transient priority |
| How the scheduler decides whether $\Pi_i$ is runnable or not | by maintaining $Q_{partition}$[i] | by maintaining $Q_{partition}$[i] | by looking up $\pi_i$ |

Fig. 12. Comparing the three partition-level schedulers

However, the refinement proof between psched and abs_psched is not straightforward. As illustrated in Fig. 12, the two differ in both the iteration order and in how they check the schedule status of a partition. To bridge the gap between them, we introduce an intermediate abstraction, int_psched (Fig. 8). Here, similar to what abs_psched does, it relies on a priority queue to iterate over all partitions from the one with the nearest deadline. On the other hand, int_psched still relies on $Q_{partition}$ to maintain the schedule status for each partition, similar to psched.

*Contextual refinement between psched and int_psched.* This involves algorithmic equivalence between the two schedulers. In particular,

- If int_psched returns $N$ (the idle partition) at time $t$, all partitions must have finished and psched will also return $N$.
- If $\Pi_i$ is scheduled by int_psched at time $t$, all higher-priority ones (having a closer deadline) must have finished. In this case, after iterating over all partitions, psched would also find that $\Pi_i$ is the highest-priority ready partition.

*Contextual refinement between int_psched and abs_psched.* We observe that the two schedulers exhibit a common structure. More specifically, they rely on the same priority queue and iterate over all partitions in the same order. The only difference is that int_psched uses the quantum value while abs_psched looks up the virtual time map to decide whether a partition has used up its budget or not.

Closing this gap between the quantum value and the virtual time map calls for the same equivalence relation as in Def. 4.1 in [Liu et al. 2020]. In particular, we prove that the following relation is preserved as the two schedulers execute so that they match each other step-by-step.

$$Q_{partition}[i] = C_i - \min\left(C_i, \pi_i(t) - \pi_i\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right)\right)$$

*Sufficiency of an individual virtual timeline.* We prove that the abstract scheduler, abs_psched, is always consistent with individual virtual timelines, such that it suffices to examine $\pi_i$ to derive $\Pi_i$'s temporal properties without considering other tasks. At any moment, at most one partition satisfies both of the following requirements: (i) the next time slot is available to it on its virtual timeline; (ii) it has not used up its budget in the current period.

This concludes the connection with the scheduler's source code. As shown in Fig. 6, this guarantees the faithfulness of the temporal abstraction such that all properties proved on top of it also hold on the system's source code.

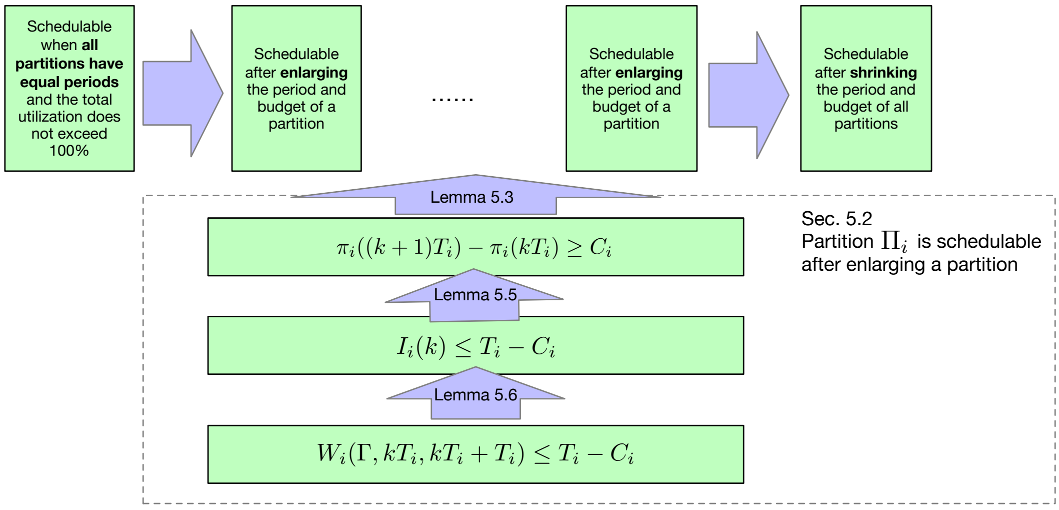

Schedulable when **all partitions have equal periods** and the total utilization does not exceed 100%
Schedulable after **enlarging** the period and budget of a partition
......
Schedulable after **enlarging** the period and budget of a partition
Schedulable after **shrinking** the period and budget of all partitions
Lemma 5.3
$\pi_i((k+1)T_i) - \pi_i(kT_i) \geq C_i$
Lemma 5.5
$I_i(k) \leq T_i - C_i$
Lemma 5.6
$W_i(\Gamma, kT_i, kT_i + T_i) \leq T_i - C_i$
Sec. 5.2
Partition $\Pi_i$ is schedulable after enlarging a partition

Fig. 13. Overview of the schedulability proof of EDF

## 5 VERIFYING AN EDF SCHEDULER

This section explains in more detail how we verify the partition-level EDF scheduler. Recall that a partition is strictly periodic and will be cut off until the next period once it has used up its budget. We formalize the proof goal as below.

*Definition 5.1 (Schedulability).* If the total utilization of all EDF partitions does not exceed 100%, for any partition $\Pi_i$, it always gets its full budget $C_i$ in each period $T_i$, i.e.,

$$\forall k \geq 0, \pi_i((k+1)T_i) - \pi_i(k * T_i) \geq C_i$$

The above schedulability property is specified on the braided virtual timeline abstraction, $\pi_i$, which describes the schedule of partition $\Pi_i$. This is a faithful abstraction since Sec. 4.2 proves the contextual refinement between psched and abs_psched, such that any property proved on $\pi_i$ carries down to the assembly code of psched.

### 5.1 Outline of the Schedulability Proof

This section outlines our schedulability proof for an EDF scheduler. The proof goal is formalized in Def. 5.1. Given the complicated construction of $\pi_i$, it is extremely challenging, if possible at all, to obtain a straightforward proof.

Rather, we reflect on the intuition behind the EDF scheduling. Its schedulability test is compositional: only the utilization, instead of the exact period or budget of a partition, matters. An immediate implication is that we can pretend that all partitions have a common period if we are only concerned about the system's schedulability. In other words, we can tweak partitions' parameters so that they share a common period,

$$C_0/T_0 \rightarrow C_0'/T, C_1/T_1 \rightarrow C_1'/T, \ldots$$

in which case the scheduling is equivalent to a weighted round-robin manner and proving the schedulability is trivial.

This insight was first discussed by Wilding [Wilding 1998], who formalized it as the optimality of the EDF and proved it in a theorem prover. However, unlike the work being presented in this paper, its proof relies on reordering an existing schedule in a sophisticated manner, making the connection with the scheduler source code challenging, if possible at all.

In this work, we follow a different path with the same insight. Instead of permitting arbitrary adjustment of partition parameters, we view it as a way of breaking down the proof obligation into smaller steps. As mentioned earlier, since the proof regarding the original partition set is challenging, we first consider a trivial case where all partitions share the same period, then adjust partition parameters one by one until we revert it to be identical to the original partition set.

Assume that the partition set $\Gamma$ contains three partitions: $\Pi_0 = (10, 40)$, $\Pi_1 = (10, 30)$, and $\Pi_2 = (20, 50)$. We define $\Omega^{EDF}(\Gamma)$ to specify that $\Gamma$ is schedulable under the EDF policy. The aforementioned proof structure is illustrated below.

*Example 5.2.* The schedulability proof for $\Gamma = \{\Pi_0, \Pi_1, \Pi_2\}$

$$\begin{aligned}
& \Omega^{EDF}(\{(150, 600), (200, 600), (240, 600)\}) \\
\Longrightarrow \quad & \Omega^{EDF}(\{(150, 600), (600, 1800), (240, 600)\}) \\
\Longrightarrow \quad & \Omega^{EDF}(\{(600, 2400), (600, 1800), (240, 600)\}) \\
\Longrightarrow \quad & \Omega^{EDF}(\{(600, 2400), (600, 1800), (1200, 3000)\}) \\
\Longrightarrow \quad & \Omega^{EDF}(\{\Pi_0, \Pi_1, \Pi_2\})
\end{aligned}$$

In the above example, we start from a partition set where all partitions have a common period, 600, while their utilizations remain the same as the original partition set. Subsequently, we *enlarge* the 2nd, 1st, and 3rd partition by a factor of 3, 4, and 5, respectively. Here, enlarging a partition by a factor of k means updating its period and budget to be k times the original value, such that the partition's utilization stays the same. Also, notice that the order in which we enlarge partitions is not arbitrary. In fact, we follow the increasing order of the original partition's period, such that at any step, the enlarged partition will have the longest period in the system. This helps us eliminate certain unnecessary cases and simplifies the proof. In the end, we *shrink* all partitions by a factor of 60 to transform them back to the original partition set.

To put it formally, for any arbitrary set $\Gamma$ containing N periodic partitions, we first compute H as the least common multiple of $\{T_0, T_1, ..., T_{N-1}\}$. Then we start from a set whose partitions share a common period H while preserving their original utilization, i.e., $\Pi_i$'s budget is $\frac{H}{T_i} C_i$. Then we enlarge partitions one by one, in increasing order of their original period. For $\Pi_i$, it is enlarged by a factor of $T_i$, with a new period $HT_i$ and new budget $HC_i$. In the end, all partitions are shrunk by a factor of H, resulting in the original set.

It is trivial to prove the schedulability of the first case, where all partitions have a common period. In the following sections, we discuss formal proofs that enlarging a partition (Lemma 5.3) and shrinking all partitions (Lemma 5.4) preserves the schedulability of the system.

Lemma 5.3 (Enlarging a partition preserves the schedulability). *Assume that the original partition set is $\Gamma$. Partition $\Pi_i \in \Gamma$ is enlarged by a factor of k, resulting in the new partition set $\Gamma'$. If $\Gamma$ is schedulable, $\Gamma'$ must also be schedulable*

Lemma 5.4 (Shrinking all partitions by a common factor preserves the schedulability). *Assume that the original partition set is $\Gamma$. Also assume there exists a common factor k, such that for any partition $\Pi_i \in \Gamma$, its period and budget are multiples of k. By dividing all partitions' periods and budgets by k, we obtain a new partition set, $\Gamma'$. If $\Gamma$ is schedulable, $\Gamma'$ must also be schedulable*

## 5.2 Enlarging a Partition

This section explains how we prove Lemma 5.3. Here, schedulability is specified using the braided virtual timeline, $\pi$, whose construction is illustrated in Fig. 9. Even though partition priorities vary

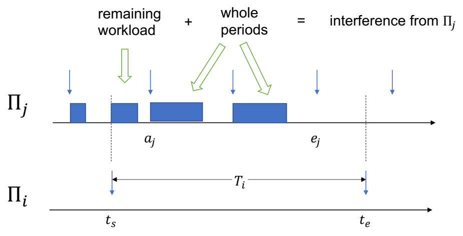


Fig. 14. Breaking down $\Pi_j$'s interference on $\Pi_i$

over time, we observe that it is possible to account for the aggregate interference between partitions in a systematic way. We define the aggregate *interference* as a ghost state in Fig. 9 Here, $I_i(\lfloor t/T_i \rfloor)$ denotes the amount of temporal interference incurred on $\Pi_i$ in the $\lfloor t/T_i \rfloor$-th period, i.e., within the physical time window $[\lfloor \frac{t}{T_i} \rfloor T_i, \lfloor \frac{t}{T_i} \rfloor T_i + T_i)$. In particular, it complements the virtual time for $\Pi_i$ in that upon any invocation of the scheduler, either its virtual time or the interference will increment by 1. This leads to the following lemma.

Lemma 5.5. *Temporal interference and virtual time complement each other*

$$\forall i, k, \pi_i((k+1) * T_i) - \pi_i(k * T_i) = T_i - I_i(k)$$

Thus, the reasoning of a partition's virtual time reduces to the reasoning of the amount of temporal interference it experiences. We further demonstrate below that this aggregate value can be broken down into portions that allow relatively straightforward computation.

*Breaking down the temporal interference.* Here, we demonstrate how the aggregate interference can be constructed under the EDF scheduling. Fig. 14 illustrates two different ways a partition $\Pi_j$ may interfere with $\Pi_i$. Here, the period of $\Pi_i$ spans from $t_s$ to $t_e$, where $t_e = t_s + T_i$. Above it, the blue blocks represent the execution of $\Pi_j$, where $a_j$ and $e_j$ denote its first and last arrival within $[t_s, t_e)$, respectively. The intuition is that any interference must come from the unfinished workload of other partitions, which happen to occupy a higher priority at the moment. We categorize them as below.

- If a period of $\Pi_j$ starts before $t_s$ but ends within $[t_s, t_e)$, as illustrated by the period that ends at $a_j$, a portion of its execution will preempt $\Pi_i$ and count toward the temporal interference. We call it the remaining workload because only the portion that is not finished before $t_s$ will interfere with $\Pi_i$. We use $wr_i(\Pi_j, \pi_j, t_s, t_e)$ to denote it since the value depends on $\Pi_j$'s remaining budget, which is easily computable if $\pi_j$ is known.
- Otherwise, if a whole period of $\Pi_j$ is within the window $[t_s, t_e)$, it will execute for its full budget before $\Pi_i$ can finish. We call it the whole period workload and denote it as $ww_i(\Pi_j, t_s, t_e)$.

In the remainder of this paper, we use $\Gamma$ to denote the set of partitions in the system. $WR_i(\Gamma, t_s, t_e)$ represents the total remaining workload from all other partitions, i.e. it equals $\sum_{j \neq i} wr_i(\Pi_j, \pi_j, t_s, t_e)$. Similarly, we let $WW_i(\Gamma, t_s, t_e)$ stand for the total whole-period workload from all other partitions. The sum of the two is denoted as $W_i(\Gamma, t_s, t_e)$, which includes all other partitions' workloads that may interfere with $\Pi_i$ within a particular window $[t_s, t_e)$.

Lastly, we prove that this accounting of workload indeed serves as an upper bound on the amount of interference. Why is it a bound instead of equaling the exact interference? Recall that we do not have a proof for the system's schedulability yet. Given such a situation, the best we can do is to

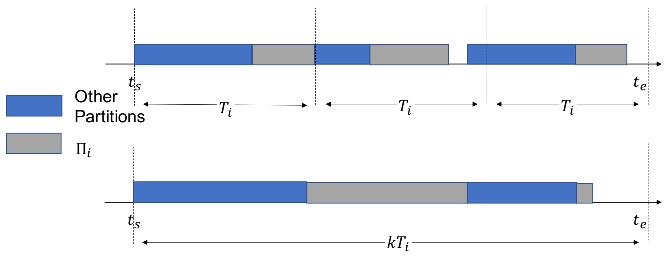


Fig. 15. Enlarging $\Pi_i$ by a factor of k

prove that the exact interference cannot go beyond the total workload, which is constrained by the budget enforcement mechanism. It requires a schedulability proof to demonstrate that the full workload will be carried out and count as interference.

Lemma 5.6. *The total workload is a faithful upper bound on the interference. For any partition $\Pi_i \in \Gamma$, and any of its period $[k * T_i, k * T_i + T_i)$,*

$$I_i(k) \le W_i(\Gamma, k * T_i, k * T_i + T_i)$$

Proof. Recall that the aggregate interference is defined iteratively as in Fig. 9. At each step t, if the time slot $[t, t+1)$ counts as interference, it must be occupied by some partition whose deadline is closer than $t + T_i$, thus included in the total workload. □

On top of it, the reasoning of braided virtual timelines reduces to the reasoning of workloads from other partitions, with formal proof that such a computation of workloads is a correct bound on a partition's temporal behavior.

Assume that the original set is $\Gamma$ and the partition $\Pi_i \in \Gamma$ is enlarged by a factor of k, resulting in the set $\Gamma'$. Given that $\Gamma$ is schedulable, we now demonstrate the proof that $\Gamma'$ is also schedulable. We consider the updated $\Pi_i$ (denoted as $\Pi_i'$) and other partitions separately.

Lemma 5.7. *The enlarged partition itself is schedulable.*

Proof. Fig. 15 illustrates the schedule of a system before (the top half) and after (the bottom half) the enlarging of $\Pi_i$. As shown in the figure, the priority of $\Pi_i'$ is effectively lowered after its enlargement, and the execution of other partitions is shifted earlier compared to the schedule before. Nevertheless, despite the variation in the actual schedule, the interference from other partitions does not increase because their aggregate workload from time 0 to $t_e$ does not change, and the portion that is finished before $t_s$ can only become bigger instead of smaller. On the other hand, the schedulability of $\Pi_i$ implies that all periods of $\Pi_i$ are schedulable within $[t_s, t_e)$, as shown in the top half of the figure. Combining the two factors, merging these periods (by enlarging the partition) will leave at least $k * C_i$ available time for $\Pi_i'$ execution. Thus $\Pi_i'$ is schedulable. □

On the other hand, an enlarged partition incurs less interference on others. The intuition is that the enlargement in effect combines multiple periods together, lowering their overall priority. We omit proof details in this paper. These two properties together prove that enlarging a partition preserves the schedulability of the system.

## 5.3 Shrinking All Partitions

Finally, we examine the last step in Example 5.2, i.e., shrinking all partitions by a common factor. Fig. 16 illustrates that the two schedules before and after the shrinking match each other by a factor

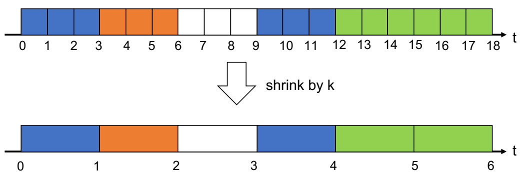


Fig. 16. Shrinking all partitions by a common factor

```
int lsched(){
  for(int j = 0; j < M; j++){
    if (index[j] != t / $p_j$){
      index[j] = t / $p_j$;
      $Q_{task}$[j] = $e_j$;
    }
  }

  int tid = M;
  for(int j = 0; j < M; j++){
    if($Q_{task}$[j] > 0){
      tid = j;
      $Q_{task}$[j]--;
      break;
    }
  }

  return tid;
}
```

Fig. 17. C code of a local (task-level) scheduler

```
int obliv_lsched(){
  for(int j = 0; j < M; j++){
    if (index[j] != lt / $lp_j$){
      index[j] = lt / $lp_j$;
      $Q_{task}$[j] = $e_j$;
    }
  }

  int tid = M;
  for(int j = 0; j < M; j++){
    if($Q_{task}$[j] > 0){
      tid = j;
      $Q_{task}$[j]--;
      break;
    }
  }

  return tid;
}
```

Fig. 18. C illustration of an oblivious local (task-level) scheduler (actually specified in Coq)

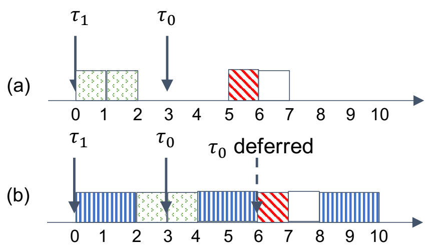
of k. It is straightforward to see that shrinking the entire partition set can be viewed as adopting a different measurement of time and should not affect the schedulability of the system. We omit proof details in this paper.

This concludes the schedulability proof of the original partition set.

Notice that this paper aims to tackle the information flow security challenge in real-time systems. As discussed in Sec. 2.2, this is a unique problem for systems that integrate potentially untrusted partitions and that schedule them in a non-static manner. For traditional real-time systems that do not suffer from information flow security vulnerabilities, they can usually adopt the same EDF scheduler (Fig. 7) to schedule real-time tasks. In this situation, temporal isolation means schedulability, i.e., each task is given its full budget within every period, regardless of the behavior of other tasks. Our framework can be easily applied to reason about the schedulability of tasks, just by replacing partition parameters in this section with task parameters.

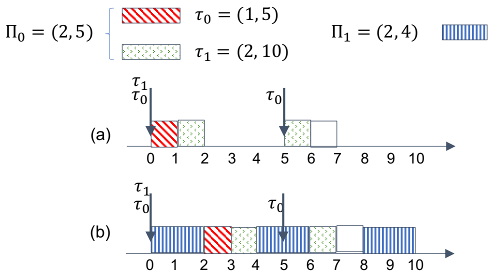
## 6 TEMPORAL ISOLATION OF THE LOCAL SCHEDULE

In this section, we explain the local scheduling within a partition and prove the temporal isolation property under the EDF-based partition scheduling.



(a) $\Pi_0$ is scheduled alone. (b) $\Pi_0$ is scheduled together with $\Pi_1$.

Fig. 19. Secured partition by deferring task arrivals



(a) $\Pi_0$ is scheduled alone. (b) $\Pi_0$ is scheduled together with $\Pi_1$.

Fig. 20. Secured partition by imposing restrictions

## 6.1 Local Scheduler within a Partition

Fig. 17 shows the local scheduler for a partition. Here, $p_i$ and $e_i$ denote the period and budget of task $\tau_i$. It uses an array `index` to track the start of new periods for each task. It also maintains an array, $Q_{task}$, to enforce tasks' budgets. At any invocation, it schedules the highest-priority task that is available to execute. Otherwise, an idle task will execute.

Notice that `lsched` relies on the global time, `t`, to maintain the index and $Q_{task}$ array for its tasks. This makes it vulnerable to interferences from other partitions. For example, as illustrated in Fig. 2, different invocation points of the enclosing partition (time 0, 1, and 5 in (a) vs. time 2, 3, and 6 in (b)) may result in a different schedule of its tasks. Such reordering may lead to illegal information flow between partitions, as described in Sec. 2.2.

## 6.2 Determinizing the Local Schedule: Restricted Tasks

As discussed in Sec. 6.1, the application of hierarchical scheduling may lead to temporal interference and further covert channels between partitions.

There are two approaches that we can take to address this vulnerability. One is to adopt a more sophisticated local schedule transformation that buffers task arrivals so that the resulting schedule is deterministic [Yoon et al. 2021]. Fig. 19 illustrates a secured partition containing the green task ($\tau_1$) and the red task ($\tau_0$). Unlike in Fig. 2, this secure partition defers the arrival of task $\tau_0$ so that it produces consistent schedules.

The other approach is to impose restrictions on task arrivals such that they are deterministic in the partition's local view despite that the partition schedule floats around (e.g., being preempted at different points in different runs). More specifically, we require that tasks' periods are multiples of the partitions' period and that tasks' arrivals are always synchronized with the boundaries of the partition's periods (i.e., 'bound' tasks [Davis and Burns 2005]). Fig. 20 demonstrates that bound tasks exhibit consistent arrival patterns regardless of other partitions. This is common among real-time applications where tasks must execute periodically and are synchronized with the base period of their enclosing partition. We take the second approach in this paper.

## 6.3 Proving Temporal Isolation of the Local Scheduler Source Code

In this section, we formally prove that the local scheduler of a partition is oblivious to other partitions if its tasks' parameters are properly restricted. Fig. 18 shows such an oblivious local scheduler. Instead of relying on the global time, `obliv_lsched` maintains its own local time, `lt`, which only increments when the enclosing partition is scheduled. Assume that the partition has

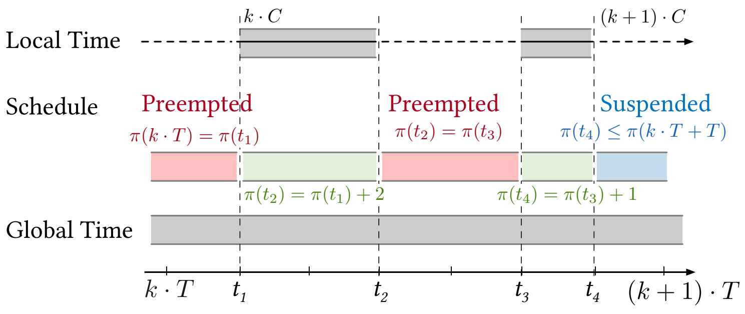


Fig. 21. Relation between the local time and global time

period T and budget C. Since `lt` increments by C when t increments by T after every partition period, `obliv_lsched` matches task' periods with `lt` by treating $lp_j = p_j * C/T$ as a task's period in the partition's local time. Since all subsequent updates on `index` and $Q_{task}$ are dependent on this local time, the resulting schedule is also solely dependent on the value of `lt`, yielding a deterministic local schedule that is oblivious to other partitions. As demonstrated in Fig. 20, in both case (a) and (b), whenever `lt` equals 0, 1, and 2, partition $\Pi_0$ will schedule $\tau_0, \tau_1, \tau_1$, respectively.

Theorem 6.1 (Temporal isolation of an EDF partition). *Consider a system with N EDF partitions, whose total utilization does not exceed 100%. For any partition $\Pi_i$, if its tasks are properly restricted, i.e. their periods $p_j$ are multiples of the partition period $T_i$, and $\Pi_i$ employs* `lsched` *(Fig. 17) for its local schedule, then its resulting local schedule is oblivious to other partitions in the system.*

*In other words, there exists an oblivious task-level scheduler,* `obliv_lsched`, *which only maintains partition-local states for its task scheduling, such that* `lsched` *must produce an equivalent schedule with* `obliv_lsched`.

This is achieved by proving the contextual refinement between `lsched` and `obliv_lsched`. A major challenge is how to formally relate the global time with the local time of a partition. Assume that the enclosing partition has period T and budget C. Informally speaking, the local time, which depicts the progress of a partition, must satisfy certain properties such as $\forall k, t = k*T \implies lt = k*C$, which is a corollary of the partition's schedulability. However, a contextual refinement proof requires a step-wise relation that is far more precise.

To formally relate the progress of a partition, depicted by its local time `lt`, with the global time, we employ braided virtual timeline for this partition, $\pi$.

Lemma 6.2 (Refinement relation between the local time and global time). *At any moment, the progress of a partition can be measured by its braided virtual timeline.*

$$lt = \lfloor \frac{t}{T} \rfloor C + min(C, \pi(t) - \pi(\lfloor \frac{t}{T} \rfloor T))$$

The intuition is illustrated in Fig. 21. A partition's schedule can be complex: it may be preempted by other partitions (red), it may shift to the highest priority and schedule for certain time slots (green), or it can exhaust the partition budget and be suspended until the next period (blue). As a comparison, all the above complexities are hidden from the partition's local time. With the help of Lemma 6.2, we prove the contextual refinement between `obliv_lsched` and `lsched`, such that the schedule based on local time is indeed a correct abstraction.

Table 1. Partition $(C_i, T_i)$ and task $(wcet, period)$ parameters for the evaluation task set. Units are in ms.

| | $\tau_{i,1}$ | $\tau_{i,2}$ | $\tau_{i,3}$ | $\tau_{i,4}$ |
|---|---|---|---|---|
| $\Pi_1$ $(2, 20)$ | $(1, 40)$ | $(2, 80)$ | $(4, 160)$ | $(7, 320)$ |
| $\Pi_2$ $(3, 30)$ | $(1, 60)$ | $(3, 120)$ | $(6, 240)$ | $(11, 480)$ |
| $\Pi_3$ $(4, 40)$ | $(2, 80)$ | $(4, 160)$ | $(8, 320)$ | $(15, 640)$ |
| $\Pi_4$ $(5, 50)$ | $(2, 100)$ | $(5, 200)$ | $(10, 400)$ | $(19, 800)$ |
| $\Pi_5$ $(6, 60)$ | $(3, 120)$ | $(6, 240)$ | $(12, 480)$ | $(23, 960)$ |
| $\Pi_6$ $(7, 70)$ | $(3, 140)$ | $(7, 280)$ | $(14, 560)$ | $(27, 1120)$ |
| $\Pi_7$ $(8, 80)$ | $(4, 160)$ | $(8, 320)$ | $(16, 640)$ | $(31, 1280)$ |
| $\Pi_8$ $(9, 90)$ | $(4, 180)$ | $(9, 360)$ | $(18, 720)$ | $(35, 1440)$ |
| $\Pi_9$ $(10, 100)$ | $(5, 200)$ | $(10, 400)$ | $(20, 800)$ | $(39, 1600)$ |
| $\Pi_{10}$ $(10, 110)$ | $(5, 220)$ | $(11, 440)$ | $(22, 880)$ | $(43, 1760)$ |

## 7 IMPLEMENTATION AND EVALUATION

This work builds on top of real-time CertiKOS [Liu et al. 2020], a single-core OS kernel featuring user-level preemption and a fixed-priority scheduler. We extend it with a hierarchical scheduler with EDF partitions and fixed-priority local schedulers (as shown in Fig. 17) in each partition. Budget enforcement is imposed on both partition- and task-level schedulers.

*Evaluation Setup.* The system is measured on an Intel NUC mini PC with an Intel Core i5-7300U processor operating at 2.6GHz (Turbo Boost is disabled) and the main memory of 8GB. The local APIC timers are configured to generate timer interrupts every millisecond, while the task and partition scheduling granularity are at every tick. This is already practical for a system with a limited number of tasks. We leave the optimization toward a more efficient task look-up and a tickless scheduling model for future work.

The evaluation consists of three parts: micro-benchmark, macro-benchmark, and temporal isolation. The first two are intended to evaluate the feasibility of using the verified scheduler in practice, while the last one is intended to show that temporal isolation is guaranteed.

*Micro-benchmark.* We first measure the cost of individual scheduler operations to reveal the overhead introduced by the hierarchical scheduler. We execute the task set shown in Table. 1 on CertiKOS for 5 minutes and collect the duration of each scheduler operation. To examine the impact of different workloads on the overheads, we vary the number of loaded partitions for multiple execution rounds.

Fig. 22-(a) presents the cost of *re-schedule*, in which the partition-level scheduler first selects a partition and then asks its local scheduler to select the next task to run. As shown in Fig. 7, we use a `while` loop to search the partition with the earliest deadline. Thus, its cost is proportional to the total number of partitions. To be noted, the cost of operations is measured in CPU cycles, which are collected by inserting `RTDSC` instructions at the beginning ($s_i$) and end ($e_i$) of each scheduler operation. We use $\sum_{i=1..N} \frac{e_i - s_i}{N} - O_\epsilon$ to compute the average operation overhead, where $N$ is the total number of operation invocations during the measurement period. $O_\epsilon$ is the performance measurement framework overhead which is calculated by instrumenting an empty function. We disabled Turbo Boost to ensure the TSC count equals the CPU cycle.

Fig. 22-(b) shows the cost of *re-schedule* that only corresponds to partition-level scheduling. It is included in the total *re-schedule* cost shown in Fig. 22-(a). Again, we choose to use a loop-based algorithm to search for the next available partition (see Fig. 7), in which the time complexity is $O(|\Pi|)$. The cost of rescheduling partitions and tasks is linearly proportional to the number of partitions. One additional partition increases the overhead of partition rescheduling by around nine cycles. A future improvement is to use a more efficient algorithm to retrieve the next partition.

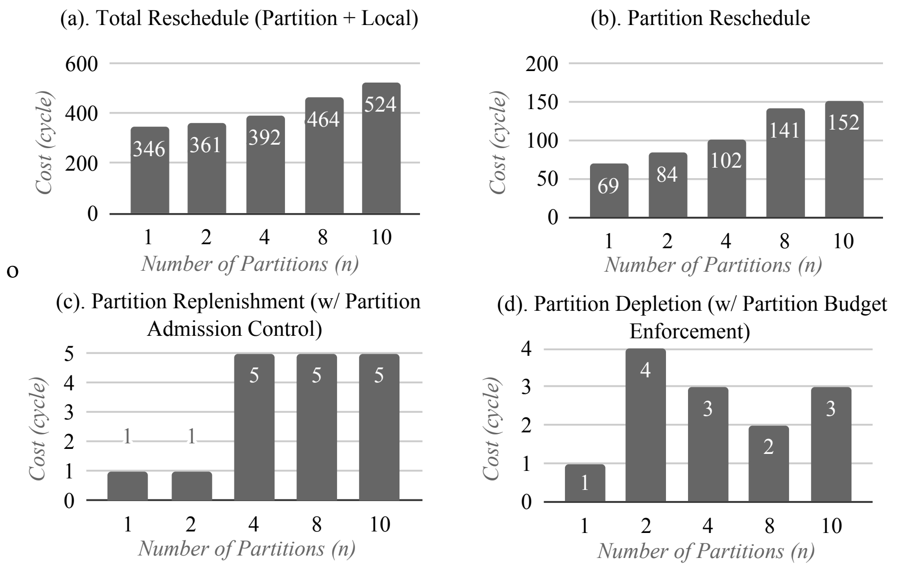


Fig. 22. Average CPU cycles of individual scheduler operations with respect to the number of partitions.

Table 2. Statistics of the difference of measured task response time and nominal response time.

| Avg | Min | Max |
|---|---|---|
| 204 ns | 113 ns | 622 ns |

Comparing the partition reschedule overhead (Fig. 22-(b)) with the whole *re-schedule* operation (Fig. 22-(a)), we can find that the partition-level scheduling does not impose a significant extra overhead.

Fig. 22-(c) shows the *partition-budget replenishment* operation, which occurs when a new period of a partition begins. The latency of such operation is minimal because this operation is essentially two instructions: an `add` instruction (of the utilization of the current scheduler workload and the admitting task) and a `comparison` (<100) instruction. Fig. 22-(d) shows the cost of *partition depletion* operation. It is called when the budget of a partition is depleted – the scheduler will suspend the partition. The costs of *partition replenishment* and *partition depletion* are both negligible and do not increase with the number of partitions.

*Macro-benchmark.* To evaluate the overall overhead introduced by the hierarchical scheduler, we use task response time measured from the moment when a task arrives to when it completes its execution. For measurements, we instrument the scheduler with a response time trace – the CPU cycle of the beginning and end time points are recorded by the scheduler. Because the response time of a task varies in the different periods due to the task/partition-level preemption, we compare the recorded response time trace with the nominal response times (i.e., analytically calculated from task parameters). We reuse the task set that is defined in Table. 1, taking Partitions 1–4 and multiplying their budgets and all the execution times by 2. We collected the response time trace for 20 seconds from CertiKOS. We define the overhead as the difference of the measured response time from the nominal one. As Table 2 shows, the overall overhead is considerably small when considering that the response times are in milliseconds (from 2 ms to about 800 ms).

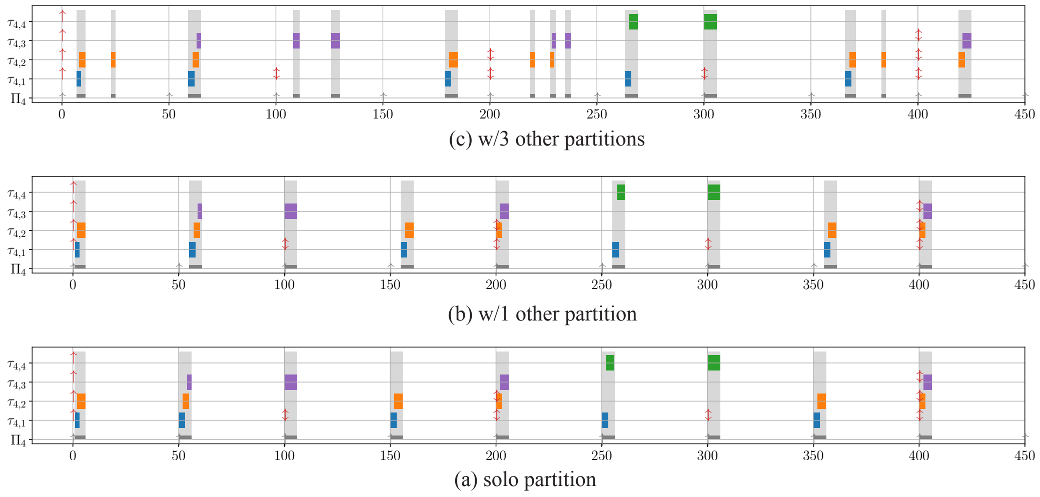


Fig. 23. Scheduling traces of partition $\{\tau_{4,1}, \tau_{4,2}, \tau_{4,3}, \tau_{4,4}\}$ running with different number of other partitions (unit: ms).

| Item | LOC in Coq |
|---|---|
| Formalization of the dynamic virtual time map and related lemmas | 2,102 |
| The schedulability proof on top of the braided virtual timelines | 16,170 |
| Functional correctness proof for the partition-level EDF scheduler's C code | 2,876 |
| Connecting the schedulability proof with the partition-level EDF scheduler | 4,876 |
| Functional correctness proof for the local task scheduler's C code | 2,963 |
| Contextual refinement proof between the local task scheduler and its oblivious abstraction | 7,594 |
| **Grand total** | 36,581 |

Fig. 24. Proof efforts over the original real-time CertiKOS

*Temporal isolation.* To demonstrate the obliviousness of a partition's local scheduling of tasks, we execute the same partition alone, with another partition, and with three other partitions, respectively. We instrument the scheduler with a set of traces, where each of them corresponds to a partition. At each schedule point, when the local_sched() function returns, we record the task ID and the global time for the current partition. We reuse the task set that is defined in Table. 1, taking the first four partitions and multiplying the budget and all execution time for each task in the first three of them by 3 (hence the total utilization is 100%).

Fig. 23 shows the scheduling traces of partition $\Pi_4$ when it runs (a) with the other three partitions, (b) with one of them, and (c) alone. Note that the exact time instants of task executions (measured in the physical time) are affected by how other partitions execute. Nevertheless, we can see that the four tasks of $\Pi_4$ follow the same local schedule (i.e., relative ordering) across all three runs.

This demonstrates the property that is stated in Theorem 6.1; that is, a partition's local behavior is indeed oblivious to other partitions. For each partition activation (after their arrival, marked as '↑'), even if it is preempted by some other partitions, e.g., at 150 ms, comparing sub-figures (a), (b), and (c), the local tasks are still scheduled the same as if no partition preemption has occurred. To be noted, Theorem 6.1 holds for all the cases; hence, we only created the single set as an example to demonstrate the property.

*Coq proof.* We prove in Coq that each partition is reserved its full budget within every period and that task scheduling is oblivious to other partitions.

Fig. 24 details the proof efforts involved in this extension. The main proof challenge of this work, i.e., the dynamic computation of time maps and the high-level reasoning on top of it, takes more than 18k LOC in Coq. On the other hand, the local task scheduler and its obliviousness to other partitions take 11k LOC in Coq.

## 8 LIMITATIONS AND FUTURE WORK

### 8.1 Kernel Overhead

Task models discussed throughout this paper assume that the kernel overhead is negligible, which is consistent with our performance evaluation result. However, non-preemptible kernel code, the interrupt handler, and the scheduler itself all consume time and eat into resources that are intended for user applications.

On the one hand, existing work reduces kernel overhead by injecting preemption points in lengthy system calls [Klein et al. 2009]. On the other hand, a formal worst-case bound is needed if the temporal reasoning takes into account kernel overhead.

### 8.2 Microarchitectural Interference

Another subtlety lies in microarchitectural-level interferences between tasks. Even though our schedulability proof guarantees a certain execution time for a user task during each period, the number of executed instructions varies if another task is sharing a common cache line and may evict previously cached entries. Mitigation techniques [Ge et al. 2019; Sha et al. 2016] mainly involve properly partitioning of shared hardware resources, ranging from cache lines to memory banks, so that interference between tasks is reduced.

These techniques are largely orthogonal to the temporal isolation proof in this work, which is specified on a logical notion of time slots, i.e. the duration between two timer interrupts. Nevertheless, mitigations of these caching effects fall into the existing reasoning framework. As discussed in [Heiser et al. 2019], we can encode these microarchitectural resources in the machine model (without concrete numbers for the execution cost) and prove that partitioning decouples the executions of different tasks.

### 8.3 Extension to Tickless Scheduling

This work adopts a tick-based scheduler, as shown in Fig. 7, which is invoked at every timer interrupt. This is a reasonable setting by itself, and we choose this design for its regularity. However, there exist more sophisticated scheduler implementations that are invoked less frequently to achieve better performance.

For example, a tickless scheduler proactively calculates the next scheduling event (e.g., starting of a new period), and sets the timer interrupt to trigger at the specified future instant (e.g., by utilizing the TSC-deadline mode of a CPU). In this way, the scheduler is only invoked when it needs to update its internal state and recalculate the task to schedule.

Our framework can be extended to accommodate such a tickless setting. The intuition is that the tick-based and tickless schedulers still follow the same scheduling policy. They only differ in their implementations, which matters for performance. Suppose that the EDF scheduler `psched` in Fig. 7 is replaced by a tickless one, `tl_psched`. We can prove that the two schedulers' states match with each other every time when `tl_psched` is invoked. And for time instants when only `psched` is invoked, its internal state does not change. In this way, we prove the contextual equivalence between the two, and further reasoning can be based on `psched` as shown in this paper.

## 9 RELATED WORK

*Mechanized Schedulability Analysis.* One of the motivations for mechanized schedulability analysis is that paper proofs are error-prone. Wilding [Wilding 1998] points out that the original paper proof for EDF in [Liu and Layland 1973] contains errors. Instead, they prove the same schedulability test for EDF using a theorem prover such that the result is machine-checkable. There are also work on the modeling and verification of real-time communication protocols [Dutertre 2000; Zhang et al. 2012] and response time analyses [Cerqueira et al. 2016; Fradet et al. 2018; Maida et al. 2022; Roux et al. 2022]. However, all of them are developed on an abstract model and their results do not apply to any scheduler source code directly.

Guo et al. [Guo et al. 2019] tackle the connection between a mechanized schedulability analysis and the scheduler's source code. However, their approach requires intrusive modifications on both sides, and it is unclear whether it is feasible for more sophisticated algorithms such as EDF partitions and their temporal isolation properties.

The recent work by Vanhems et al. [2022] features a mechanized proof for an EDF scheduler implementation. They adopt an abstraction-and-refinement approach to bridge the gap between the high-level reasoning and the concrete scheduler implementation. However, unlike our work, their proof does not connect to a user-facing criterion, e.g., the total utilization of all partitions does not exceed 100%. Instead, they build their proof on top of the assumption that within any arbitrary time interval, the total budget of jobs does not exceed the length of that interval.

A more fundamental difference is in the design goal of the work, which leads to different ways of formalizing scheduling behaviors. Vanhems et al. [2022] was only designed for schedulability of tasks but not for information flow security of partitions, which is all that is expected of traditional real-time systems. As a result, it formalizes schedulability as the absence of deadline miss. On the contrary, this paper aims to prove the confidentiality of partitions. This requires our formalization to be expressive enough to specify the scheduling behavior of both partitions and their tasks. We use braided virtual timeline to faithfully characterize temporal behaviors of individual partitions. Thanks to its expressiveness, we first use this formalization to prove schedulability, which is a property of braided virtual timelines. We then leverage this property to reason about the behavior of a single partition and prove that its local schedule of tasks is not affected by other partitions.

*Formal Verification of Real-Time OS Kernels.* Most of existing work on verified real-time OS kernels [Andronick et al. 2016; Xu et al. 2016] do not address end-user real-time guarantees, i.e. schedulability of components.

Liu et al. [Liu et al. 2020] propose the virtual timeline framework to address temporal isolation between components. One of their main contributions is a structured way of lifting the reasoning into an abstract domain while guaranteeing the faithfulness of this abstraction. In particular, they leverage the fixed hierarchy among tasks to compute the temporal behavior of each of them in decreasing order of priority. This enables a straightforward static construction of the abstraction, called virtual time maps, but is also a limitation since it is tightly coupled with the fixed-priority scheduling. It cannot extend to other scheduling algorithms directly.

On the contrary, this work proposes a generic abstraction that can describe dynamic-priority scheduling and connect with the scheduler's source code. In particular, it proposes an innovative use of a priority queue to make the precedence relation between tasks explicit, and further constructs time maps dynamically on top of it. This is a significant difference from [Liu et al. 2020] and it reflects a generic structure among preemptive scheduling. For example, fixed-priority scheduling can be easily characterized as a special case where the priority queue remains constant.

*OS Kernels and Isolation.* OS Kernels have long served the role of managing physical resources and ensuring isolation between different components. While the formal verification of spatial isolation has been tackled by the literature [Murray et al. 2013; Nelson et al. 2017], they do not provide as strong temporal isolation guarantees. For example, both works rely on the static allocation of time slots to establish temporal isolation between components, which is too restrictive. Liu et al. [Liu et al. 2020] verify an OS kernel with both temporal and spatial isolation, but they are limited to the fixed-priority scheduling. As a comparison, our work finds a generic way of handling any preemptive scheduling algorithm and proving related temporal isolation properties.

On the implementation side, Ge et al. [Ge et al. 2019] mitigate micro-architectural-level interferences by making those hardware resources explicit and properly partitioning them among independent tasks. Heiser et al. [Heiser et al. 2019] discuss possible approaches toward its formal verification.

Lyons et al. [Lyons et al. 2018] design a capability-based interface to allow users to reserve CPU bandwidth for critical tasks. Their underlying algorithm is based on sporadic servers. As a comparison, our work can accommodate similar reservation mechanisms (e.g., polling servers) while also providing a formal isolation guarantee. In particular, the EDF algorithm we prove allows us to manage CPU utilization in a compositional way.

## 10 CONCLUSIONS

This paper addresses dynamic-priority partitions with temporal isolation. In particular, hierarchical scheduling has be historically adopted by centrally developed systems, where all components are trusted and thus there is no security concern. However, with the increasing trend of accommodating multiple real-time applications from different vendors on a single processor, such a scheme may lead to covert channels. We propose using budget-enforcing EDF partitions and imposing restrictions on tasks within a partition to achieve a balance between flexibility, high-utilization, and temporal isolation.

In tackling the challenges with formally verifying EDF partitions, we adopt a dynamic construction of virtual timelines to lift the reasoning of a concrete scheduler into an abstract domain. This is a significant innovation since EDF scheduling results in braided timelines that intersect and shift priorities and are never properly layered on each other. On top of it, we prove that an EDF partition always receives its full budget and that its local task schedule is oblivious to the other partitions. These entail temporal isolation between partitions.

## ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This material is based upon work supported in part by NSF grants 1945541, 2019285, and 2118851, and by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of their employers or sponsors.

## REFERENCES

L. Abeni and G. Buttazzo. 1998. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium.* 4–13. https://doi.org/10.1109/REAL.1998.739726

Aeronautical Radio, Inc. 2010. *Avionics Application Software Standard Interface: ARINC Specification 653P1-3.* Aeronautical Radio, Inc.

June Andronick, Corey Lewis, Daniel Matichuk, Carroll Morgan, and Christine Rizkallah. 2016. Proof of OS Scheduling Behavior in the Presence of Interrupt-Induced Concurrency. In *Proceedings of 7th International Conference on Interactive Theorem Proving (ITP).* Springer International Publishing, Nancy, France, 52–68. https://doi.org/10.1007/978-3-319-43144-4_4

F. Cerqueira, F. Stutz, and B. B. Brandenburg. 2016. PROSA: A Case for Readable Mechanized Schedulability Analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS).* 273–284. https://doi.org/10.1109/ECRTS.2016.28

David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-end Verification of Information-flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16).* ACM, New York, NY, USA, 648–664. https://doi.org/10.1145/2908080.2908100

R. I. Davis and A. Burns. 2005. Hierarchical Fixed Priority Pre-Emptive Scheduling. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05).* IEEE Computer Society, Washington, DC, USA, 389–398. https://doi.org/10.1109/RTSS.2005.25

Z. Deng, J.W.-S. Liu, and J. Sun. 1997. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings Ninth Euromicro Workshop on Real Time Systems.* 191–199. https://doi.org/10.1109/EMWRTS.1997.613785

B. Dutertre. 2000. Formal analysis of the priority ceiling protocol. In *Proceedings 21st IEEE Real-Time Systems Symposium (RTSS'00).* IEEE Computer Society, Washington, DC, 151–160. https://doi.org/10.1109/REAL.2000.896005

Pascal Fradet, Maxime Lesourd, Jean-François Monin, and Sophie Quinton. 2018. A Generic Coq Proof of Typical Worst-Case Analysis. In *2018 IEEE Real-Time Systems Symposium (RTSS).* 218–229. https://doi.org/10.1109/RTSS.2018.00039

Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys'19).* ACM, New York, NY, USA, Article 1, 17 pages. https://doi.org/10.1145/3302424.3303976

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15).* ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building Certified Concurrent OS Kernels. *Commun. ACM* 62, 10 (sep 2019), 89–99. https://doi.org/10.1145/3356903

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16).* USENIX Association, Savannah, GA, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

Xiaojie Guo, Maxime Lesourd, Mengqi Liu, Lionel Rieg, and Zhong Shao. 2019. Integrating Formal Schedulability Analysis into a Verified OS Kernel. In *Computer Aided Verification - 31st International Conference (CAV'19), July 15-18, Proceedings.* Springer, Berlin, Heidelberg, 496–514. https://doi.org/10.1007/978-3-030-25543-5_28

Gernot Heiser. 2020. The seL4 Microkernel – An Introduction (White paper). Revision 1.2. (June 2020).

Gernot Heiser, Gerwin Klein, and Toby Murray. 2019. Can We Prove Time Protection?. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19).* Association for Computing Machinery, New York, NY, USA, 23–29. https://doi.org/10.1145/3317550.3321431

Mathai Joseph and Paritosh Pandya. 1986. Finding response times in a real-time system. *Comput. J.* 29, 5 (1986), 390–395. https://doi.org/10.1093/comjnl/29.5.390

Jung-Eun Kim, Tarek Abdelzaher, and Lui Sha. 2015. Budgeted generalized rate monotonic analysis for the partitioned, yet globally scheduled uniprocessor model. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium.* 221–231. https://doi.org/10.1109/RTAS.2015.7108445

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09).* Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

J. Lehoczky, L. Sha, and Y. Ding. 1989. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *[1989] Proceedings. Real-Time Systems Symposium.* 166–171. https://doi.org/10.1109/REAL.1989.63567

C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61. https://doi.org/10.1145/321738.321743

Jane W. S. W. Liu. 2000. *Real-Time Systems* (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. 2020. Virtual Timeline: A Formal Abstraction for Verifying Preemptive Schedulers with Temporal Isolation. *Proc. ACM Program. Lang.* 4, POPL (Jan. 2020), 31 pages. https://doi.org/10.1145/3371088

LynxOS-178 2022. *LynxOS-178*. https://www.lynx.com/products/lynxos-178-do-178c-certified-posix-rtos.

Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. 2018. Scheduling-Context Capabilities: A Principled, Light-Weight Operating-System Mechanism for Managing Time. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 26, 16 pages. https://doi.org/10.1145/3190508.3190539

Marco Maida, Sergey Bozhko, and Björn B. Brandenburg. 2022. Foundational Response-Time Analysis as Explainable Evidence of Timeliness. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 231)*, Martina Maggio (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 19:1–19:25. https://doi.org/10.4230/LIPIcs.ECRTS.2022.19

Toby C. Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From General Purpose to a Proof of Information Flow Enforcement. In *2013 IEEE Symposium on Security and Privacy (SP'13), Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, Washington, DC, 415–429. https://doi.org/10.1109/SP.2013.35

Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17), Shanghai, China, October 28-31, 2017* (Shanghai, China). ACM, New York, NY, USA, 252–269. https://doi.org/10.1145/3132747.3132748

Pierre Roux, Sophie Quinton, and Marc Boyer. 2022. A Formal Link Between Response Time Analysis and Network Calculus. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 231)*, Martina Maggio (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:22. https://doi.org/10.4230/LIPIcs.ECRTS.2022.5

L. Sha, M. Caccamo, R. Mancuso, J. E. Kim, M. K. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford. 2016. Real-Time Computing on Multicore Processors. *Computer* 49, 9 (Sept 2016), 69–77. https://doi.org/10.1109/MC.2016.271

Insik Shin and Insup Lee. 2003. Periodic resource model for compositional real-time guarantees. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003.* 2–13. https://doi.org/10.1109/REAL.2003.1253249

Brinkley Sprunt, Lui Sha, and John Lehoczky. 1989. Aperiodic task scheduling for Hard-Real-Time systems. *Journal of Real-Time Systems* 1 (1989), 27–60. https://doi.org/10.1007/BF02341920

Florian Vanhems, Vlad Rusu, David Nowak, and Gilles Grimaud. 2022. A Formal Correctness Proof for an EDF Scheduler Implementation. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 281–292. https://doi.org/10.1109/RTAS54340.2022.00030

VxWorks 653 2022. *Wind River VxWorks 653 Platform*. https://www.windriver.com/products/vxworks/certification-profiles/#vxworks_653.

Matthew Wilding. 1998. A machine-checked proof of the optimality of a real-time scheduling policy. In *Computer Aided Verification*, Alan J. Hu and Moshe Y. Vardi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–378. https://doi.org/10.1007/BFb0028759

Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A Practical Verification Framework for Preemptive OS Kernels. In *Computer Aided Verification: 28th International Conference (CAV'16), Toronto, ON, Canada, July 17-23, 2016, Proceedings*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Berlin, Heidelberg, 59–79. https://doi.org/10.1007/978-3-319-41540-6_4

Man-Ki Yoon, Mengqi Liu, Hao Chen, Jung-Eun Kim, and Zhong Shao. 2021. Blinder: Partition-Oblivious Hierarchical Scheduling. In *Proc. 30th USENIX Security Symposium on (USENIX Security 2021), August 2021.*

Xingyuan Zhang, Christian Urban, and Chunhan Wu. 2012. Priority Inheritance Protocol Proved Correct. In *Interactive Theorem Proving (ITP'12)*. Springer, Berlin, Heidelberg, 217–232. https://doi.org/10.1007/978-3-642-32347-8_15