SF3P: A Framework to Explore and Prototype Hierarchical Compositions of Real-Time Schedulers

Andres Gomez*†, Lars Schor†, Pratyush Kumar†, and Lothar Thiele†
*Integrated Systems Laboratory, ETH Zurich, 8092 Zurich, Switzerland
†Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
gomeza@iis.ee.ethz.ch firstname.lastname@tik.ee.ethz.ch

***Abstract*—The trend to integrate multiple functionalities on the same (off-the-shelf) hardware has made the selection of the right scheduling algorithm and configuration difficult. This selection requires the designer to validate any scheduling decision already during early design steps on the target architecture, e.g., by using a reconfigurable scheduling framework running in the user-space. In this paper, we first identify the requirements that such a scheduling framework must fulfill. Then, we propose SF3P: an open-source framework that meets these requirements. To this end, we define an interface common to all scheduling algorithms and separate the scheduling algorithm from its low-level implementation. With these features, SF3P can not only prototype a scheduler at high level of abstraction, but also execute the implemented task-set on specific hardware. Furthermore, SF3P can hierarchically compose scheduling algorithms, useful in the mixed criticality domain, and could also be used to explore different scheduling policies in the system optimization phase. We demonstrate these features by implementing SF3P on top of a POSIX-compliant operating system on two different platforms: Raspberry Pi and an Intel Core i7 desktop system.**

I. Introduction

Scheduling theory is a well-studied topic in the real-time system literature. Focusing on the logic and theoretical proof of scheduling algorithms, they provide the application designer with a large design space. To provide timing guarantees, one must carefully select the types of algorithms such as event-based, time-triggered, preemptive and non-preemptive and their parameters, such as priorities, slot sizes, budgets etc. Theorical works often give little evidence of which scheduling algorithm to choose and how to configure the selected scheduling algorithm for a given application set. Even more, when targeting commodity setups, i.e., systems with off-the-shelf hardware and software components, timing guarantees will depend on the operating system and the computer architecture. These dependencies are often ignored in literature, even though they might affect the selection of the scheduling algorithm. Recent design trends show a move from federated to integrated architectures in which multiple applications are being executed on the same computing platform [1]. Scheduling algorithms must be able to temporally isolate different applications and offer perfomance guarantees, making the processes of selecting the right scheduling algorithm even more complex.

In this work, we argue that choosing the right scheduling algorithm and configuration for commodity setups requires the application designer to begin validating all selections in the early design steps on the target architecture, e.g., by prototyping different candidate algorithms. These requirements can be met by prototyping candidate schedulers on different hardware platforms using a high-level framework. To be useful and efficient, such a framework must

1) offer high-level building blocks, i.e., a classical scheduler library,
2) allow the design of custom schedulers with minimal effort,
3) enable the hierarchical composition of schedulers, and
4) have minimal requirements from the underlying software and hardware components to increase compatibility.

Conventional implementations of scheduling algorithms either modify the base of a standard kernel to provide support for new scheduling techniques [2]–[5] or add new functionality in the form of dynamically loaded kernel-space modules [6], [7]. This methodology is difficult to maintain and places strict requirements on the underlying HW/SW platform and does not fulfill the previously stated requirements for fast prototyping schedulers.

This paper presents the Scheduling Framework For Fast Prototyping[1] (SF3P), which is an open-source framework that meets the above discussed requirements by a unique combination of key features. First, SF3P enables the comparison and verification of a large set of configurations of the same scheduling algorithm by separating the configuration from the implementation of the algorithm itself.Second, SF3P includes a library of commonly used scheduling algorithms and can be easily extended with custom schedulers. This is possible because SF3P introduces a user-space scheduling layer with clearly defined interface to low-level components. Third, SF3P allows many meaningful hierarchical compositions of scheduling algorithms with multiple levels. This is possible because of two design choices. SF3P abstracts scheduling algorithms and streams of jobs by an entity called runnable. It also defines an interface between runnables called criteria, based upon which the scheduling algorithms make their scheduling decisions. Finally, SF3P has wide compatibility because it resides in the user-space and has minimal interaction with the kernel-space. We demonstrate the effectiveness of SF3P by implementing it on top of POSIX-compliant operating systems and targeting two different platforms, namely an ARM-based Raspberry Pi [8] and an Intel Core i7 desktop system. For a large set of task-sets, we measured the performance of the framework in terms of the difference between theoretical and observed schedulability. We found this difference to be less than 1% for the desktop processor. Finally, we show the advantages of the proposed framework, and the need for hierarchical scheduling, by discussing a flight management software example from our industrial partners.

II. Related Work

A wide variety of scheduling software has been proposed to solve the resource allocation challenges that arrived with new scheduling algorithms and hardware platforms. In the following, we discuss the closest scheduling software by broadly classifying them into kernel patches/modules, user-space/hierarchical frameworks.

We refer to any modification to a standard kernel's resource allocation software for supporting for new scheduling techniques as a kernel patch. The LITMUS^{RT} [2] patch, the SCHED_ DEADLINE kernel patch [4], and the AQuoSA framework [5] are examples of such kernel patches. All of them have in common that they are able to provide high timing requirements for a small subset of scheduling algorithms and platforms. Extending the proposed concepts to new

[1] http://www.tik.ee.ethz.ch/~euretile/scheduling

scheduling algorithms or operating systems could be costly since it requires re-verification and re-testing of the kernel.

Instead of patching a kernel, new functionalities can also be added in the form of dynamically loaded kernel-space modules. The Hijack framework [6] intercepts system calls and hardware interrupts in order to enforce real-time policies for user-space thread execution. A similar approach is taken by the ExSched framework [7], which allows user-space schedulers to be loaded as plugins to a kernel module through a unified interface. Compared to our approach, these frameworks support scheduling among multiple applications instantiated as separate processes. However, their downside is that they run, at least partly, in the kernel-space and could affect other functionalities of the system and increase the instability of the system. Furthermore, porting (or creating) kernel modules to new platforms is typically more expensive than porting a user-space library.

By following a similar approach, user-space frameworks are the most relevant to our work. They use a standard kernel without additional modules and add a scheduling layer in the user-space. The meta-scheduler framework described in [9] provides a portable middleware layer component to implement real-time scheduling algorithms. `wuthreads` [10] is a user-space threading library that supports the hierarchical composition of schedulers by having a top-level scheduler that assigns a global priority to each loaded low-level scheduler. A user-space library implementing the clustered Earliest Deadline First (EDF) algorithm is described in [11]. The authors investigate the development of a user-space library for multi-core systems, but do not state the costs to implement scheduling algorithms other than the clustered EDF algorithm. Finally, the authors of [12] show that complex mixed-criticality algorithms can be implemented in the user-space with low overhead.

Frameworks that enable more than one scheduling policy to be enforced hierarchically have first been proposed in [13] by considering two-level hierarchical schemas. The HLS framework proposed in [14] provides the ability to compose multiple soft real-time schedulers by having a predefined hierarchical structure with a fixed priority (FP) scheduler on top. In [15], a composition method to derive timing requirements for a hierarchy of schedulers supporting EDF and FP schedulers is described. While these works focus mostly on providing timing requirements for hierarchical scheduling frameworks, we focus on the challenges that arrive when practically implementing a hierarchy of scheduling algorithms. The ExSched framework [7] also supports 2-level hierarchies of FP and EDF schedulers. In contrast to all previous works, SF3P permits the comparison, in terms of performance and overhead, as well as the hierarchical composition of arbitrary real-time schedulers over a wide variety of hardware and software platforms.

III. Background

Scheduling, i.e., the process of assigning resources to a workload according to specific policies, can be characterized by a system model $M = (R, A, W)$ consisting of a resource model R, a scheduling algorithm A, and a workload model W. In this section, we will first discuss the workload model W. Afterwards, we will summarize some of the most common scheduling algorithms.

The basic unit of the considered workload model W is a *task* τ Each instantiation of a task τ is called a *job*. Upon arrival, a job is said to be *eligible* for being scheduled. We consider periodic tasks where jobs are characterized by an average-case execution time C_{avg}, a worst-case execution time C_{wc}, and a period P. A job might have an associated relative deadline D, which defines the maximum allowed time between its release and finish times. We call a task-set T feasible under a given scheduling algorithm A if the algorithm will always schedule every job $\tau \in T$ such that all deadlines are met.

A scheduling algorithm's role is to select a job to run from a set of eligible jobs according to predefined criteria at the appropriate times. Scheduling algorithms may be broadly classified as time-triggered or event-triggered algorithms. Event-triggered scheduling can be further differentiated as either priority-based or not. Lastly, priority-based algorithms can have either static (assigned per task) or dynamic (assigned per job) priorities. An additional distinction between algorithms is whether they are preemptive, meaning that jobs which are being executed can be interrupted before they have finished. The concepts adopted in the proposed scheduling framework are valid for static- and dynamic-priority algorithms, as well as non-preemptive and preemptive scheduling algorithms. We will focus on four classical scheduling algorithms, namely Earliest Deadline First (EDF), Fixed Priority (FP), First-in-First-Out (FIFO), and Time Division Multiple Access (TDMA).

We also consider valid hierarchical compositions of schedulers. Such a composition can be represented as a tree where the top-level node represents the shared resource, the leaf nodes represent a workload, and all other nodes represent a specific scheduling algorithm. Fig. 1 illustrates the scheduling model of a hierarchical composition of three scheduling algorithms: EDF, FIFO, and FP. This configuration will be explained in detail in Section V-D.

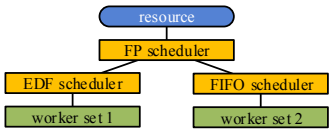
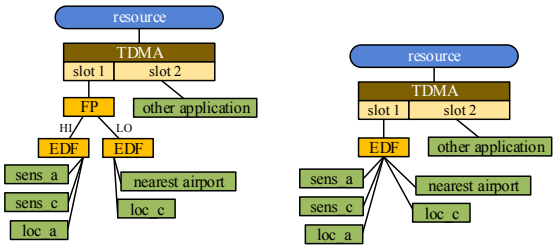


Fig. 1. Hierarchical Scheduling System with EDF, FP, and FIFO.

IV. Challenges and Design Approach

In this section, we present the challenges and design approach of SF3P through a real-world example sourced from our industrial partners. Flight Management Software (FMS) is responsible for several on-board functions on an airplane. In particular, it reads different sensors, fuses the sensor data, and computes different parameters to localize the airplane's position. The localization data is used for multiple purposes, such as planning the flight trajectory and computing the nearest airport. We study a subset of these tasks which belong to five task-groups named *sens_a*, *sens_c*, *loc_a*, *loc_c*, and *nearest_airport*. Each task-groups also has a criticality. According to the official manuals, tasks belonging to *sens_a*, *sens_c* and *loc_a* have criticality level B, while tasks belonging to *loc_c* and *nearest_airport* have a lower criticality level D. As first proposed in [16], under nominal conditions all jobs must meet their deadlines. Under some exceptional conditions, jobs of tasks with lower criticality level must start to miss deadlines before affecting jobs of higher criticality levels.

To meet the requirements of the mixed-criticality task-set, one may begin by proposing the scheduler configuration shown in Fig. 2a. The resource is partitioned into TDMA slots, one of which is assigned to the FMS application. Within this slot, a fixed priority scheduler is used to prioritize tasks of the higher criticality-level B. Within each priority, an EDF scheduler is used to maximize the schedulability of the tasks of that priority. Similarly, one may also propose the configuration shown in Fig. 2b, where only one EDF scheduler is used to schedule all of the FMS tasks. At the outset, it is not clear how to implement these configurations on top of a standard operating system nor how much of the design can be re-used for one configuration to the other. We aim to solve these problems with SF3P.



(a) FMS with configuration C1. (b) FMS with configuration C2.

Fig. 2. Examples of Hierarchical Configurations

The goal of this work is to provide the system designer with a tool that enables the fast evaluation of scheduling algorithms on specific hardware platforms. As discussed with the flight management software, this exploration also includes hierarchical compositions of different schedulers. There are two key challenges in the design of SF3P. The first challenge, i.e., the hierarchical composition of scheduling algorithms, is met by introducing two abstract concepts, namely runnables and criteria. A runnable abstracts the external interface of a scheduling algorithm and a concrete workload in a sense that it enables a scheduling algorithm to perform its decisions based on a set of runnables instead of a set of eligible jobs. In addition, every runnable has an attached criteria encapsulating the parameters that might be used by a scheduling algorithm. We will detail these abstract concepts in the next section. Note that not all compositions of scheduling algorithms might make sense. For instance, in [17], it is argued that schedulers with the strictest timing requirements should be scheduled near the top of the hierarchy. We account for these restrictions by allowing time-sharing scheduling algorithms to be only scheduled below other time-sharing scheduling algorithms.

We do not focus on deriving timing guarantees for different hierarchical configurations. Neither do we claim that our proof-of-concept implementation achieves real-time performance on any (general-purpose) operating system. Instead, our primary focus is on how to decouple the high-level functional description of the (hierarchical) schedulers from the system-dependent low-level implementation. We envisage that such a framework will complement the understanding of schedulers, particularly in the mixed criticality domain, where experimental evidence can precede (currently unknown) theoretical results.

V. Abstract Scheduling Model

In this section, we present an abstract model of a scheduling framework that consists of a set of hierarchically organized scheduling algorithms. By abstracting the scheduling model from the actual implementation, we aim to achieve flexibility in the sense that scheduling algorithms can be (almost) arbitrarily connected and new scheduling algorithms can be added without significant costs. We tackle this problem in two steps. First, a unified interface is introduced enabling the arbitrary composition of scheduling algorithms. Afterwards, the operation semantics of a workload and a scheduling algorithm are described based on the introduced interface.

A. Abstract Scheduling Entity

As shown in Section IV, the hierarchical composition of scheduling algorithms with multiple levels is an indispensable feature of modern scheduling systems. However, current scheduling frameworks are mostly focused on two hierarchical levels, require kernel-space extensions, or are inflexible in the sense that they only support predefined compositions of schedulers. In contrast, modern scheduling systems must be flexible, e.g., a scheduler must be able to simultaneously schedule other schedulers and tasks. We achieve this flexibility in our framework by first defining a unified interface that abstracts the external interfaces of a scheduler and a workload.

TABLE I. Events exchanged between two runnables.

Event	Sender	Receiver	Description
$activate$	parent	child	child obtains access to shared resource.
$deactivate$	parent	child	child loses access to shared resource.
$ready$	child	parent	child becomes eligible to run, i.e., it has an eligible job to execute.
$finished$	child	parent	child is no longer eligible to run, i.e., it has no more eligible jobs to execute.
$update$	child	parent	child's criteria has changed (e.g., by a new job arrival). The parent might have to re-evaluate its scheduling decision.

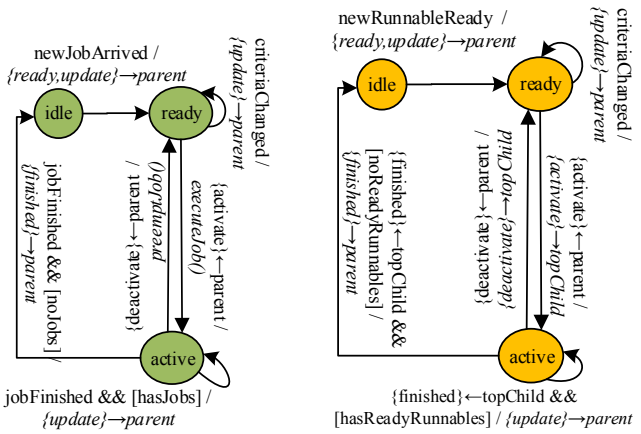
We call an entity that might be scheduled by a scheduler a *runnable*. A runnable r represents either a *scheduler* or a *worker* whereby a worker is a collection of eligible jobs of a certain task. In other words, there exists exactly one worker per task. The scheduling model can then be represented by a tree of runnables whereby each leaf node represents a worker and all other nodes represent schedulers. A runnable r has an attached *criteria* c, which is an encapsulation of parameters that might be used by a parent to prioritize amongst its children. The parameters might include a (static) *priority*, a *period*, an *arrival time*, and an *absolute deadline*. Having the notion of a criteria enables the scheduler to schedule other runnables based on their criteria, thus to equally handle schedulers and tasks. A runnable can ask for the criteria of its children and send the events $activate$ and $deactivate$ to the children. Similarly, a runnable can send the events $ready$, $finished$, and $update$ to its parent, see Table I for a description of the events.

B. Operation Semantics

After having specified a unified interface between two runnables, next, we discuss the operation semantics of a runnable by differentiating between worker and real-time scheduler. Due to space constraints, we will only discuss event-triggered schedulers. By having an abstract operation semantics, we aim to isolate the functional description of the scheduling model from the actual implementation, enabling a predictive behavior of the scheduling system independent of the implementation details.

Worker. Fig. 3a sketches the operation semantics of a worker. The worker is in the *idle* state if it has no eligible jobs, in the *ready* state if it has eligible jobs but has not been activated, and in the *active* state if it has been activated (by the parent). The worker can start executing the next eligible job by calling function `executeJob` and preempt the execution of the current executing job by calling function `preemptJob`.

Real-Time Scheduler. A real-time scheduler s operates on two lists of runnables. Runnables in the *ready* list are eligible to run and sorted according to predefined parameters of the runnables' criteria. For instance, in case of FP, the parameter of interest is the *priority*, while in case of a FIFO scheduler it is the *arrival time*. The top element points to the runnable that should be activated in case the scheduler is activated and the criteria of scheduler s points to the criteria of the top element. The *idle* list contains all runnables that are assigned to scheduler s, but not eligible to run. The operation semantics of a (preemptive) real-time scheduler is outlined in Fig. 3b. Similar to a worker, the scheduler is in the *idle* state if every child is in the *idle* state, in *ready* state if at least one child is in a *ready* state, and in the *active* state if any child is in the *ready* state and the scheduler has been activated. In case the scheduler is non-preemptive, the scheduler



(a) **Worker:** $e \rightarrow r$ refers to sending event e to recipient r. $e \leftarrow s$ refers to receiving event e from sender s.

(b) **Scheduler:** $e \rightarrow r$ refers to sending event e to recipient r. $e \leftarrow s$ to receiving event e from sender s, and `topChild` to the top element of the *ready* list.

Fig. 3. Operational Semantics for Worker and Scheduler entities.

would only be able to switch from *active* to *ready* state when the current job has finished its execution.

C. Correctness

We will now describe the basic properties to be satisfied in order to faithfully represent the above discussed semantics. We will later show that our implementation fulfills these criteria. The use of the abstract entity runnable enables us to list these properties inductively at the unit of a scheduler as follows.

1) Whenever a scheduler is in active or ready state, then its criteria must be equal to the criteria of the head of its *ready* list, which has been prioritized according to the scheduler's parameters.
2) A scheduler can be in the idle state only when all child runnables are in the idle state.
3) When a scheduler is in the ready state, at least one of the child runnables must be in the ready state.
4) When a scheduler is in the active state, exactly one of the child runnables must be in the active state.

D. Hierarchical Example

In order to illustrate the previously described operation semantics, we revisit the scheduling model outlined in Fig. 1 and analyze it in detail. The scheduling system consists of a FP scheduler that serves an EDF and a FIFO scheduler. Each worker set consists of an unspecified number of workers. A sample execution of the considered scheduling system is outlined in Fig. 4. First, an eligible job arrives at a worker from set 2, which causes all involved schedulers to re-evaluate their criteria. Afterwards, the FP scheduler selects the FIFO scheduler to be activated, which then activates its child worker, which executes its job. The execution is interrupted by another job that arrives at worker from set 1, which again causes a re-evaluation of the criteria of the EDF and FP schedulers. After deactivating the FIFO scheduler, the FP scheduler activates the EDF scheduler, which then activates the worker from set 1. Finally, after the worker from set 1 has finished, the FP scheduler will re-activate the worker from set 2, in order for it to complete its job.

VI. Portable Implementation with POSIX

Following the goal to isolate the functional description of the scheduling framework from the actual implementation, we described in the previous section the abstract operation semantics of a scheduler

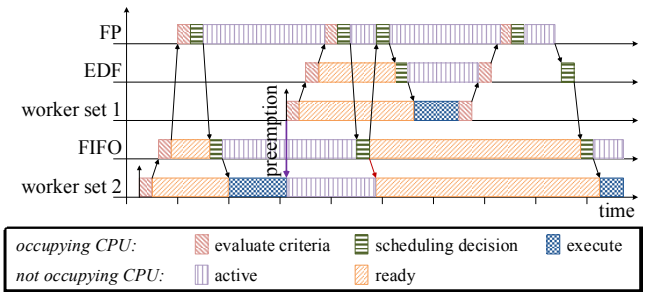


Fig. 4. Sample execution a system sketched in Fig. 1. For readability, the length of the *execute* segments has been shortened.

and a worker. The proposed operation semantics might be implemented in different ways depending on the demands of the system. In this section, we discuss a specific implementation of the framework using the API specified by the POSIX standard. The POSIX standard is supported by a wide variety of operating systems including many variants of Unix and RTOS.

The overall goal of this work is to explore and compare different (hierarchical) scheduling models in an efficient manner. In this context, efficiency means exploring the different design options in a fast, safe, and reasonably accurate manner. We argue that a user-space library is the only viable choice if the designer wants to compare various scheduling models in early design steps. When testing and comparing new scheduling algorithm prototypes, the main concern is the relative performance of different schedulers amongst each other, not the absolute performance. Even though the kernel-space can provide high timing guarantees, it might not be feasible for prototying schedulers before the specific software and hardware environments have been fully specified. Protyping schedulers in the user-space has low requirements and offers greater interoperability.

In order to implement a scheduling framework in the user-space, an operating system must provide support for three functionalities. First, the concurrent execution of multiple entities, as provided by a concurrency manager. Second, the concurrency manager must be able to schedule the individual entities according to their priority, which is dynamic. Finally, timers are required to support time-triggered schedulers. Minimally, the operating system must support preemptive priority scheduling and the `pthread_*` functions.

Given a POSIX-compliant operating system, we implement SF3P as a set of interacting threads with different priorities and manipulate the priorities to select the worker to execute. A priority-based (kernel-level) scheduler will then select the highest-priority unblocked thread for execution.

Each worker and scheduler is implemented as a separate POSIX thread. To handle job arrivals, an additional entity called dispatcher is introduced. The dispatcher is a separate POSIX thread that communicates with the workers upon the arrival of a job. The priority of each scheduler is tied to its hierarchical level. Suppose that the system has N levels of schedulers, then the top-level scheduler receives priority L_1, all schedulers of the second-level receive priority L_2, until all schedulers of the N-th level receive priority L_N with[2] $L_1 > L_2 > \ldots > L_N$. In addition, the dispatcher has priority L_{N+1}, the active worker has priority L_{N+2} and all other workers have priority L_{N+3}, again with $L_N > L_{N+1} > L_{N+2} > L_{N+3}$, see Fig. 5. Whenever a scheduler finishes its role of updating the criteria and signaling the parent and child runnables, the scheduler thread blocks itself. This allows the lower-priority worker threads to

[2]We use the "$>$" to denote the "higher-priority-than" relationship.

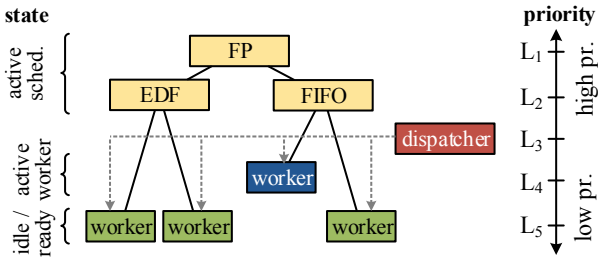


Fig. 5. Priority management enables the hierarchical composition of scheduling algorithms.

execute in-between scheduling decisions.

We like to think of the basic operation principle in two phases: a bottom-up criteria update phase, and a top-down activation phase. At the start, all runnables are in the idle state. Then, the dispatcher receives a job from a specific worker, which moves into the ready state and thus signals to its parent runnable. The runnable updates the criteria and moves into the ready state. This recursive process continues until the top most runnable. Subsequent to this bottom-to-top criteria update phase, the top most runnable activates its selected child runnable, moving the child from the ready state to the active state. This recursive process continues from parent to child till the worker with the new job is reached. After this top-down activation phase, this worker is moved into the active state, and its priority raised to L_{N+2}. Finally, the worker starts to execute its new job.

Now, let a new job of a different worker arrive. The dispatcher, having a higher priority, preempts the executing worker, and begins the recursive criteria update. If the top most runnable selects the new worker, it must first deactivate the previous worker before activating the new one. With these features, we can assert that our implementation can faithfully model the operation semantics presented in the previous section.

It should be noted that many POSIX-compliant OS's also implement fair scheduling algorithms. However, these can be bypassed with root privileges, giving the developer unrestricted access to setting thread priorities and controlling their execution.

Correctness. We will now briefly describe how the above implementation ensures the correctness properties described in Section V-C. Whenever a new job arrives, the dispatcher thread ensures that the appropriate runnables move to state ready if they are idle and update their criteria if necessary. The higher priority of the dispatcher thread allows it to preempt any active worker to register a new job. This bottom-up criteria update phase implemented by each runnable satisfies Properties 1 to 3. Whenever the top-most scheduler decides to change the currently active worker, one scheduler at each level of the hierarchy activates exactly one of its child runnables, satisfying Property 4.

VII. Experimental Evaluation

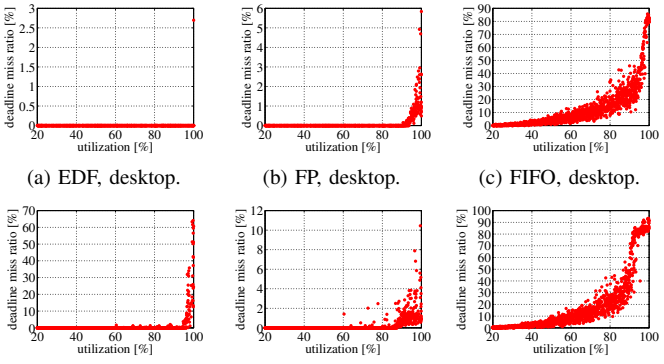
In this section, we evaluate the performance of the proposed framework. The goal is to answer the following questions. *a*) How much does the overhead introduced by the proposed framework distort the behavior of a scheduling algorithm? *b*) How is the overhead affected by the individual components of the scheduling framework? *c*) How much does the overhead increase with the number of hierarchical levels? To answer these questions, we evaluate the performance of various task-sets on two different target platforms. Finally, we revisit the flight managemen system and evaluate the impact of different scheduling configurations on the performance metrics.

Setup. In order to demonstrate the portability of the SF3P, we use two testing environments. The first is a desktop environment, consisting of an eight-core, 3.4 GHz Intel i7 running the 3.5.0-17-generic Linux Kernel. The second, an embedded environment, is a Raspberry Pi (RPI) model B rev 2 with a 700 MHz ARM V6 processor running the 3.6.11 kernel. During all evaluations, the operating system's run-level is lowered to 1 so that only essential system services were running, and the framework's CPU affinity is set to one processor.

If not specified otherwise, the used compiler is G++ 4.7.2 with optimization level O2 for the desktop platform, and G++ 4.6.3 again with optimization level O2 for the Raspberry Pi. Furthermore, the simulation time is set to 10 seconds in all experiments.

Scheduler Performance. The goal is to quantify how much the overhead introduced by the proposed scheduling implementation distorts the behavior of the scheduling algorithm. In particular, we measure the deadline miss ratio, i.e., the fraction of jobs which miss their deadlines, and compare it to known theoretical results for EDF and Rate Monotonic, a particular FP algorithm with priorities tied to task periods. We generate 1500 random task-sets. The number of tasks per task-set is uniformly distributed from 5 to 50 and a task is randomly classified as either a short or a long task. Execution times are uniformly distributed from 5 ms to 10 ms for short tasks and from 40 ms to 50 ms for long tasks. The periods of the tasks are set so that the system utilization is uniformly distributed from 20 % to 100 %. This choice of task parameters ensures that the expected number of job arrivals per unit time is the same for all task-sets. The relative deadlines of tasks are assumed to be equal to the period of the task. Afterwards, the task-set is scheduled under EDF, FP, and FIFO both on the desktop platform and the Raspberry Pi.

For each generated task-set, Fig. 6 shows the deadline miss ratio as a function of the utilization for both platforms. For the desktop computer, the highest utilization at which a deadline is missed for EDF is 100 %. Task-sets with utilization of even 99.9 % meet all their deadlines. This indicates that SF3P is very close to the theoretical bound of 100% [18]. However, there is a larger number of deadline misses on the Raspberry Pi. The worse performance on the Raspberry Pi was expected, among other things, because of the much higher cost of context switching. However, when the experiment is repeated with the periods and execution times scaled by a factor of 10, which lowers the relative weight of context switching, the results closely resemble that of the desktop.



(a) EDF, desktop. (b) FP, desktop. (c) FIFO, desktop.

(d) EDF, Raspberry Pi. (e) FP, Raspberry Pi. (f) FIFO, Raspberry Pi.

Fig. 6. Deadline miss ratio for three schedulers and two platforms.

The extensive results of Fig. 6 allow us to compare scheduling algorithms for soft real-time systems. For instance, it is interesting to observe the trend of the deadline miss ratio for the FIFO scheduler as

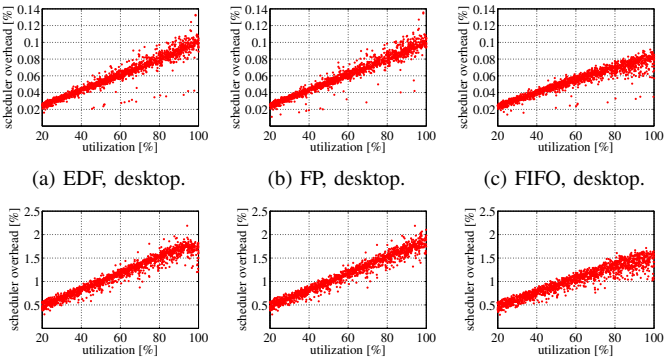
a function of the utilization, which, to the best of our knowledge, has not been theoretically investigated. If deadline miss ratios up to 10 % can be tolerated, then a utilization of around 60 % can be supported by FIFO on the desktop computer.

Overall, for the appropriate range of task-set parameters, SF3P does not distort the performance of the scheduler algorithms, and thus enables comparing different choices. This confirms our claim that a user-space library is a viable choice to compare various scheduling models in early design stages.

Scheduler Overhead. Next, we quantify the overhead introduced by the scheduling framework for various flat scheduling algorithms. The role of SF3P is to execute worker threads. Anything else can be considered as overhead. This overhead includes the registration of new jobs, their insertion into the eligible queue, and enforcing priority changes in case of a preemption or job completion. We classify these components into two categories: *scheduler* and *non-scheduler* overhead. The scheduler overhead is the time taken to register new jobs at the appropriate positions in the eligible task-queues. Depending on the complexity of the scheduling algorithm and the number of tasks, this overhead could be either large or small. The scheduler overhead is measured using the POSIX-CPU-timers for the scheduler and dispatcher threads, normalized over the total run-time. We use the same 1500 task-sets as in the previous experiment. Fig. 7 shows the measured scheduler overhead plotted against the overall system utilization, for both platforms and for the three scheduling algorithms. The following observations can be made:

1) *Utilization-dependence*: For all cases, the trend of the overhead increases linearly with the utilization.
2) *Scheduler-dependence*: For either platform, all algorithms have about the same overhead for the smallest utilization of 20 %. The increase in overhead with utilization is slower for FIFO than for EDF and FP. Indeed, for the highest utilization, the overhead of FIFO is up to 20 % less than that of EDF and FP.
3) *Platform-dependence*: The overhead is much higher (around 25 ×) on the Raspberry Pi platform than on the desktop platform.

The non-scheduler overhead is the time taken by the lower-level services and the operating system during the simulation. This overhead might include constant background processes of the operating system and the time to translate user-space decisions by SF3P to the underlying concurrency manager. The non-scheduler overhead may not directly be related to the design of SF3P, the task-set, or the scheduling algorithm. We measure the non-scheduler overhead by subtracting the total run-time from the sum of the POSIX-CPU-timers for all POSIX threads in SF3P, normalized over the total run-time.



(a) EDF, desktop. (b) FP, desktop. (c) FIFO, desktop.

(d) EDF, Raspberry Pi. (e) FP, Raspberry Pi. (f) FIFO, Raspberry Pi.

Fig. 7. Scheduler overhead for three schedulers and two platforms.

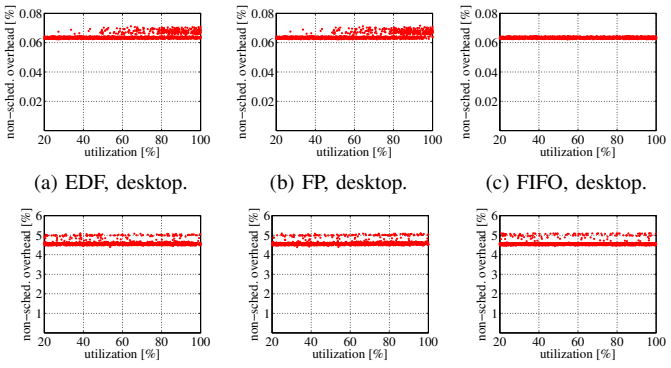
For the same task-sets, the non-scheduler overhead is plotted in Fig. 8. The following observations can be made:

1) *Utilization- and scheduler-dependence*: For either platform, the non-scheduler overhead remains fairly constant across different values of the utilization and the scheduling algorithm.
2) *Platform-dependence*: The non-scheduler overhead is much higher (around 75 ×) on the Raspberry Pi platform than the desktop platform.

Overhead of Multiple Hierarchy Levels. We now study the effect of different levels of scheduler hierarchy on the overhead of the framework. To this end, we generate 1500 task-sets, each with 32 tasks whose execution times are uniformly distributed from $10ms$ to $40ms$ and the utilization from 50% to 90%. These tasks are executed in different EDF configurations which vary from 1 to 5 hierarchy levels. One level of hierarchy means that all tasks are scheduled under a single EDF scheduler. For two levels of hierarchy, the top-level scheduler is EDF with two child EDF schedulers. Each of these child schedulers has 16 tasks. The other configurations are similarly created. Even though all configurations are equivalent in terms of their functionality, they will have different overheads. We execute all five configurations on both platforms and measured the scheduler and non-scheduler overheads, see Fig. 9. On both platforms, the scheduler overhead increases linearly with the levels of the hierarchy. However, for five levels of hierarchy, the overhead may still be considered acceptable: about 0.15 % on the desktop and 2 % on the Raspberry Pi. On the other hand, the non-scheduler overhead remains fairly constant for the different hierarchy levels. Indeed, this overhead is about the same as we measured for the very different task-set in Fig. 8.

Hierarchical Scheduling for Multiple Criticality Levels. Consider again the Flight Management Software (FMS) introduced in Section IV. We implemented a subset of 11 tasks of the application which are classified into two criticality levels: a higher level B and a lower level D. We profiled the tasks on the desktop computer. The difference between the tasks' average- and worst-case execution times, C_{avg} and C_{wc}, varies by a factor of 1.02 to 4, depending on the task. The FMS application is isolated from another co-executing application A which has an average utilization of 85%. If all jobs execute for C_{avg}, all deadlines will be met. However, if they all execute for C_{wc}, some deadlines will be missed. For this scenario, we compare three different scheduler configurations: (C1) the hierarchical scheduler configuration with three levels shown in Fig. 2a, (C2) a hierarchical scheduler configuration with two levels shown in Fig. 2b, and (C3) a flat EDF scheduler.

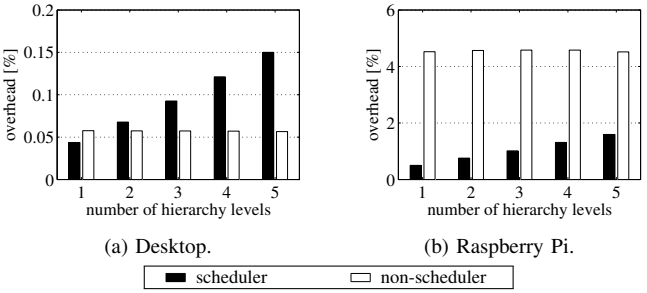
All three configurations were run on SF3P and the results can be



(a) EDF, desktop. (b) FP, desktop. (c) FIFO, desktop.

(d) EDF, Raspberry Pi. (e) FP, Raspberry Pi. (f) FIFO, Raspberry Pi.

Fig. 8. Non-scheduler overhead for three schedulers and two platforms.



(a) Desktop. (b) Raspberry Pi.

Fig. 9. Overhead of a hierarchical composition of EDF schedulers.

seen in Table II. It should be noted that only configurations C1 and C2 preserve the timing isolation across applications, since application A has no deadline misses. Furthermore, C1 prioritizes higher criticality tasks, as evidenced by the 0% deadlines misses of criticality level B. In terms of performance, the overhead rises with increasing levels of hierarchy, a clear trade-off between performance and temporal isolation and applications *and* task-groups.

TABLE II. COMPARISON OF DIFFERENT CONFIGURATIONS FOR THE WORST-CASE EXECUTION OF FMS.

Config.	Deadline Miss Rate [%]			Total Scheduling Overhead [%]
	Crit. level B	Crit. level D	Application A	
C1	0	28	0	0.71
C2	22	23	0	0.44
C3	5	7	12	0.38

A typical OS would only natively support a configuration like C3, which clearly does not meet the requirements of mixed-criticality scheduling. This highlights the need for a scheduling framework with a hierarchical composition of schedulers.

Lessons Learned. The experimental results provide several key insights. SF3P faithfully reproduces expected scheduler performance for the appropriately chosen range of task-set parameters. Indeed, this range is more conservative for the resource-constrained Raspberry Pi. The decomposition of the overhead into scheduler and non-scheduler components helps to separate two different aspects of the framework. The scheduler overhead helps to select the right scheduler algorithm and to identify the utilization that can be supported. In particular, FIFO has a lower overhead than EDF and FP, but in all cases the overhead increases linearly with utilization. The non-scheduler overhead, which is independent of the scheduling algorithm, the utilization, and levels of hierarchy, can be an important factor during system optimization. When going from the desktop computer to the Raspberry Pi, the scheduler overhead increases about $25\times$ while the non-scheduler overhead increases about $75\times$. On the Raspberry Pi, the non-scheduler overhead is dominant. Thus, to improve real-time performance on embedded platforms, it is necessary to optimize the kernel mechanisms like context switches and interrupts.

VIII. CONCLUSIONS

In this paper, we presented the Scheduling Framework For Fast Prototyping (SF3P), an open-source framework for fast prototyping of complex scheduling algorithms. Recent design trends as, for instance, the integration of multiple functionalities on the same hardware, are supported by the ability to hierarchically compose scheduling algorithms to execute tasks of multiple criticality levels. We define a unified interface between scheduling algorithms and separate the scheduling algorithm from the low-level implementation. SF3P is designed to be in the user-space with minimal interface to the kernel-space enabling a large variety of target platforms to be supported. In particular, the effectiveness of SF3P is finally demonstrated by implementing it on top of POSIX-compliant operating systems on two different platforms, namely Raspberry Pi and an Intel Core i7 desktop processor.

IX. ACKNOWLEDGMENTS

This work was supported by EU FP7 project EURETILE, under grant number 247846.

REFERENCES

[1] H. Kopetz, "From a Federated to an Integrated Architecture for Dependable Embedded Systems," DTIC Document, Tech. Rep., 2004.

[2] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "$LITMUS^{RT}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *Proc. RTSS*, 2006, pp. 111–126.

[3] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta, "An Efficient and Scalable Implementation of Global EDF in Linux," in *Proc. OSPERT*, 2011.

[4] D. Faggioli, M. Trimarchi, and F. Checconi, "An Implementation of the Earliest Deadline First Algorithm in Linux," in *Proc. SAC*, 2009, pp. 1984–1989.

[5] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA-adaptive Quality of Service Architecture," *Software: Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.

[6] G. Farmer and R. West, "Hijack: Taking Control of COTS Systems for Real-Time User-Level Services," in *Proc. RTAS*, 2007, pp. 133–146.

[7] M. Asberg, T. Nolte, S. Kato, and R. Rajkumar, "ExSched: An External CPU Scheduler Framework for Real-Time Systems," in *Proc. RTCSA*, 2012, pp. 240–249.

[8] "Raspberry Pi Website," http://www.raspberrypi.org.

[9] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 613–629, 2004.

[10] F. Kuhns, "wuthreads: Implementing a User-space Threading Library," http://www.arl.wustl.edu/~fredk/Courses/OS/wuthreads.html, 2005.

[11] M. S. Mollison and J. H. Anderson, "Bringing Theory Into Practice: A Userspace Library for Multicore Real-Time Scheduling," in *Proc. RTAS*, 2013, pp. 283–292.

[12] H.-M. Huang, C. Gill, and C. Lu, "Implementation and Evaluation of Mixed-Criticality Scheduling Approaches for Periodic Tasks," in *Proc. RTAS*, 2012, pp. 23–32.

[13] Z. Deng and J. W. S. Liu, "Scheduling Real-Time Applications in an Open Environment," in *Proc. RTSS*, 1997, pp. 308–319.

[14] J. Regehr and J. A. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in *Proc. RTSS*. IEEE, 2001, pp. 3–14.

[15] I. Shin and I. Lee, "Periodic Resource Model for Compositional Real-Time Guarantees," in *Proc. RTSS*, 2003, pp. 2–13.

[16] S. Vestal, "Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance," in *Proc. RTSS*, 2007, pp. 239–243.

[17] J. Regehr, J. Stankovic, and M. Humphrey, "The Case for Hierarchical Schedulers with Performance Guarantees," University of Virginia, Tech. Rep. CS-2000-07, 2000.

[18] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.