YASMIN: a Real-time Middleware for COTS Heterogeneous Platforms

Benjamin Rouxel
b.rouxel@uva.nl
University of Amsterdam
Netherlands

Sebastian Altmeyer
altmeyer@informatik.uni-augsburg.de
Augsburg University
Germany

Clemens Grelck
c.grelck@uva.nl
University of Amsterdam
Netherlands

ABSTRACT

Commercial-off-the-shelf (COTS) heterogeneous platforms provide immense computational power, but are difficult to program and to correctly use when real-time requirements come into play: A sound configuration of the operating system scheduler is needed, and a suitable mapping of tasks to computing units must be determined. Flawed designs lead to sub-optimal system configurations and, thus, to wasted resources or even to deadline misses and system failures.

We propose *YASMIN*, a middleware to schedule end-user applications with real-time requirements in user space and on behalf of the operating system. *YASMIN* combines an easy-to-use programming interface with portability across a wide range of architectures. It treats heterogeneity on COTS embedded platforms as a first-class citizen: YASMIN supports multiple functionally equivalent task implementations with distinct extra-functional behaviour. This enables the system designer to quickly explore different scheduling policies and task-to-core mappings, and thus, to improve overall system performance.

In this paper, we present the design and implementation of *YASMIN* and provide an analysis of the scheduling overhead on an Odroid-XU4 platform. We demonstrate the merits of *YASMIN* on an industrial use-case involving a search-and-rescue drone.

CCS CONCEPTS

• **Computer systems organization** → **Real-time operating systems**; *Embedded software*; • **Software and its engineering** → Software libraries and repositories.

KEYWORDS

Middleware, Real-Time Systems Deployment, Scheduling Deployment

ACM Reference Format:
Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. 2021. YASMIN: a Real-time Middleware for COTS Heterogeneous Platforms. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3464298.3493402

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in:
Middleware '21, December 6–10, 2021, Virtual Event, Canada
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8534-3
https://doi.org/10.1145/3464298.3493402

1 INTRODUCTION

Commercial-off-the-shelf (COTS) heterogeneous parallel platforms are very popular as they offer (in this hardware segment) unprecedented computational power at low cost. They typically combine a potentially heterogeneous multi-core CPU (e.g. ARM big.LITTLE) with a powerful GPU and, possibly, various additional hardware accelerators [27].

A large segment of embedded computing must meet real-time constraints while not being safety-critical, e.g. Internet-of-Things (IoT) [39] or edge computing [37], including a range of cyber-physical systems (CPS) [8]. Supporting real-time applications targeting such platforms is insanely complex as all required analyses need to be adapted for each type of execution unit (core, accelerator, etc.). Moreover, heterogeneous parallel architectures immediately create a complex scheduling and mapping problem of application tasks to execution units with highly different timing and energy properties. Furthermore, guaranteeing real-time properties on such systems after deploying these applications is often a nightmare as the execution environment is mostly constrained by vendor-provided (or vendor-adapted) operating systems (OS), which most often lack support for real-time techniques as exhibited by the research community.

To enforce timing properties when deploying real-time applications, designers have the choice to use a modified kernel, e.g. *Litmus^RT* [7], or real-time patches for general-purpose OS [5]. However, these solutions preclude using specific hardware drivers used in vendor-specific OS setups and, therefore, are often of limited use in practice. For instance, accelerators might not be usable without proprietary drivers. In fact, COTS embedded heterogeneous platforms are mostly bound to specific OS versions. For example, the Apalis TK1 [38] board can only run a Linux v3.10 kernel. Likewise, proprietary device drivers, mainly for the embedded GPU and further accelerators, set tight limits to change or modify the OS, not to mention that kernel modifications are cumbersome, difficult, error-prone and non-portable.

We propose a novel middleware: YASMIN (*Yet Another Scheduling MIddleware for exploratioN*) that facilitates the deployment of real-time applications on heterogeneous COTS platforms running on top of a COTS OS. Compared to previous real-time-related attempts (see Akesson et al. [4] for a recent survey), YASMIN is, to the best of our knowledge, the first middleware to embrace heterogeneity as a central design concern.

The survey by Akesson et al. further shows that industrial practitioners are very keen on using COTS OS in conjunction with libraries to deploy real-time systems. Hence, YASMIN precisely matches industrial needs:

- *Customisation*: YASMIN is highly customisable through clear separation of concerns (mapping, scheduling, priority ordering ...). Thus, adding a new state-of-the-art technique to YASMIN is much simpler and faster than adding it into an OS kernel such as Linux.
- *Adaptability*: Through support for multiple versions of tasks YASMIN permits users to change the effective behaviour of the application at run-time to address evolving environmental characteristics, such as the detection of faults or cyber-attacks, low battery status, etc.
- *Maintainability*: YASMIN is not dependent on any specific OS or OS version. Hence, upgrading a system to benefit from security patches or bug fixes is considerably easier using YASMIN than with a deployment environment bound to a specific kernel, e.g. [3].
- *Portability*: YASMIN runs on top of any POSIX-compliant OS and is not bound to any specific platform. Therefore, executing an application compiled with YASMIN on different platforms merely requires recompilation.
- *Compatibility*: When a specific platform has no RTOS support, e.g. due to vendor-specific drivers, YASMIN provides more timing guarantees than what a vanilla OS has on offer: soft real-time guarantees can be given on a vanilla Linux [32].
- *Flexibility*: Using YASMIN with different workload packages supports different configurations for each package, such as different task models, scheduling strategies, etc. This property is shown by the aforementioned survey [4] to be a prerequisite to deploy industrial systems.
- *Design space exploration*: Deciding which scheduling policy is the best for a system is rarely trivial. YASMIN offers multiple scheduling options, which can be switched at compile time. Hence, RT-experts and non-experts alike can explore the scheduling design space to select the technique that delivers best performance.

At the time of writing YASMIN supports the following scheduling policies: global earliest deadline first (G-EDF), partitioned earliest deadline first (P-EDF), global fixed priority (G-FP) and partitioned fixed priority (P-FP). Furthermore, YASMIN supports the following priority assignments: deadline-monotonic (DM), rate-monotonic (RM) or user-defined (UDF).

The range of scheduling policies might not seem particularly exotic or broad, but in our view this is an orthogonal issue: adding further scheduling policies to YASMIN would first and foremost be an engineering effort, not a research effort. The design of YASMIN, both concerning the API and the implementation, very much facilitate extension in this direction. More importantly, the proposed design of a versatile, extendable and portable user space middleware immediately shows its merits: scheduling policies are implemented once in the library and immediately become available to all application programs making use of YASMIN.

YASMIN is part of a more comprehensive endeavour to facilitate rapid prototyping and deployment of non-safety-critical real-time applications targeting heterogeneous parallel COTS platforms. Application components, their functional interplay, timing properties and requirements can be specified in a high-level coordination DSL [30]. Following a mostly automated generative approach [32], our compiler tool chain turns a high-level description of an application into C code ready for binary code generation by a target-specific C compiler. The whole tool chain, including YASMIN, is available under a GPLv3 licence [2].

The remainder of this paper is organised as follows: In Section 2 we discuss fundamental assumptions such as the underlying task model. In Section 3 we present design and implementation of YASMIN and elaborate on the various options and design choices we support. We empirically validate YASMIN in Section 4, and we apply it on an industrial use-case in Section 5. We review related work in Section 6 and draw conclusions in Section 7.

2 APPLICATION MODEL

We consider non-safety-critical real-time systems composed of a set of sporadic or periodic tasks. Each task represents an indivisible (or atomic) feature of the end-user application. The minimal time interval T (or period) separating two consecutive task activations must be provided to our scheduler. In addition, we allow non-periodic tasks managed by the end-user where no regular pattern can be given to the scheduler. Real-time tasks must complete their execution before a deadline D relative to the period. We support all three common deadline schemes: implicit ($D = T$), constrained ($D \leq T$) and arbitrary to the period.

To embrace heterogeneity we adopt recent task models representing each task with a set of versions [30] or variants [22]. All versions of a single task are functionally equivalent and expose the same interface (i.e. inputs, outputs), but each one has its own distinct non-functional behaviour, i.e. worst-case execution time (WCET), energy consumption, etc.

The immediate motivation for multi-version tasks lies in the scheduling and mapping complexity induced by heterogeneous platforms, where it is commonly not a-priori decidable which tasks should exclusively run on the CPU and which should exclusively run on one (or more) of the various accelerators. Consider an application with two tasks A, B, and each task has two versions: one running 100% on the CPU and one running 1% of the time on the CPU and 99% on the GPU. These two tasks are independent and have the same period. Hence, they could potentially run in parallel. On the target platform, however, only a single GPU is available. Hence, the two GPU versions of A and B cannot execute in parallel. However, the presence of different versions allows us to run the GPU version of A at the same time as the CPU version of B or vice versa. This would be beneficial as long as the CPU version does not take longer than running the two GPU versions one after the other. We empirically demonstrated in [30] that deciding which version to execute at each task instance is not straightforward. This question is rather part of the scheduling problem, and it is common that depending on global circumstances and objectives, the same task may sometimes preferably be executed on the CPU and in other cases on the GPU, see [30] for details.

The versatility of multi-version tasks goes beyond the above. The computing unit heterogeneity may exhibit different ISAs per core. In the presence of generic ISA compatibility multi-version tasks can still provide task implementations particularly optimised

for execution on a specific type of execution unit. Furthermore, application designers could easily play with implementation variants that expose different non-functional behaviour (e.g. energy, time, security) and let YASMIN automatically select the best suited one in a certain context and under concrete objectives.

YASMIN further supports tasks with precedence constraints, so-called *directed acyclic (task-)graphs (DAG)*. Other graph-based task models, such as Synchronous DataFlow (SDF) [23], must *a-priori* be transformed (or expanded for SDF) to comply with a DAG task model. Each edge in a graph represents a causal dependence between two tasks. This may be a data dependency, or it can be used to prevent side-effects between them. The source node of an edge must complete its execution before activating the sink. As in most graph-based task models, YASMIN supports activation patterns and relative deadlines described at the graph level: The whole graph is considered either sporadic or periodic. Only the root node(s) (which have no predecessors) is activated at an activation event triggering all subsequent nodes while the leaf node(s) (which have no successors) must complete before the deadline.

3 YASMIN DESIGN & IMPLEMENTATION

We design YASMIN as a library to be compiled individually and linked to the end-user program. YASMIN is highly modular and allows (1) the use of various scheduling policies and, (2) easy switching between them at compile time using a configuration header file.

We implement YASMIN in structured C-code following real-time and MISRA-C 2012[1] coding guidelines to enable the use of WCET analysis tools, such as AbsInt's aiT [11] or Heptane [21]. We systematically refrain from using dynamic memory allocation, and loops are statically bounded. To accomplish this, we make use of C-header configurations to define constants used throughout the library, e.g. the number of threads or the number of tasks.

YASMIN is compatible with any POSIX-compliant OS. However, we do rely on the *pthread_set_affinity_np* non-POSIX function that binds a thread to a specific core. Similar requirements can be found in previous work [26, 33].

3.1 YASMIN API

The library is configured at compile time using a configuration file. In this file, pre-processor definitions set, among others, the type of scheduling, the type of mapping and the priority assignment. Each different scheduling strategy requires different mandatory information to perform adequately, but we keep a uniform interface for all scheduling configurations. The configuration is applied to the whole compiled binary, only one scheduling policy is allowed at a time. In order to switch to another policy, the application must be recompiled with new parameters.

Table 1 presents the API of YASMIN. All functions are prefixed with `yas_`. For conciseness we omit the prefix throughout the paper. The API is common to all scheduling strategies. This allows for easy switching at compile time without modifications of the user code.

[1]We checked for MISRA-C compliance using the trial version of PC Lint Plus [36]

Table 1: Full API of YASMIN

API	Description
struct TData { char *name, u64 period, u64 deadline, u16 virt_core_id, u64 release_offset}	Structure to describe a task. Some fields are optional depending on the configured scheduling policy.
void init(void)	Initialise YASMIN.
void cleanup(void)	Wait for all worker threads to finish and close.
bool start(void)	Start to execute the tasks.
void stop(void)	Stop pushing new tasks into the ready queue. All tasks already pushed will be executed.
TID task_decl(TData *d)	Declare a task to the scheduler.
void task_activate(TID t)	Activate a non-recurring task for immediate schedule.
VID version_decl(TID t, FuncPtr f, void *f_static_args, VSelect props)	Add a version to the task with user specific properties.
HID hwaccel_decl(char *name)	Declare a hardware accelerator
void hwaccel_use(TID t, VID v, HID a)	Declare a hardware accelerator used by a task version.
channel_decl(CID, datatype, size)	Macro to declare a channel of type *type* identified by *CID* containing *size* items of type *datatype*.
channel_connect(TID src, TID dst, CID)	Macro to connect a source and a destination task using the specified channel identified by *CID*.
channel_push(CID, datatype d)	Macro to push a value of type *datatype* in the FIFO identified by *CID*. To be used in user function body.
channel_pop(CID, datatype *d)	Macro to pop a value of type *datatype* in the FIFO identified by *CID*. To be used in user function body.

The end-user program must first call the *init* function that initialises different structures of our library. Then, the user must declare the various tasks using *task_decl* and their associated versions with *version_decl*.

YASMIN supports DAG-based tasks. We provide a mechanism to declare and manage FIFO channels required between causally dependent tasks within a DAG. The pre-processor macro *channel_decl* defines the FIFO channel buffer. Connecting two tasks to

use this channel is done with *channel_connect*. The channel can be accessed from within user tasks with the *channel_push* and *channel_pop* functions to push data to and to pop data from a channel, respectively.

Hardware accelerators can be declared with *hwaccel_decl* and linked to a task version with *hwaccel_use*. Our scheduler is, therefore, aware of accelerator usage and can apply smart strategies to select a specific version at runtime according to selected criteria, see Section 3.2 for more details.

At this stage no user code has yet been executed, and no scheduling has been performed. It is only after the call to the *start* function that the scheduler starts to run the application. Calling the *stop* function stops the scheduler. Then, either the main program performs the finalisation of the application with *cleanup*, or the schedule can be resumed with a new call to *start*. It is only possible to alter the task set while the schedule is not running, hence enabling multi-mode scheduling [15]. For conciseness we omit all functions to alter the task set in the following API tables.

3.2 Heterogeneity & Multi-Version

With embedded platforms hardware accelerators are usually a scarce resource: today's system typically come with no more than a single GPU. If multiple tasks need to access an accelerator then they might need to wait for the resource to become available. To avoid this form of congestion we introduce multi-version tasks. A task may have one implementation targeting the GPU, another one using some other accelerator and yet another one targeting the CPU. Since accelerator usage is declared to our scheduler using the API call *hwaccel_use*, it can detect that the targeted accelerator is busy and that it might be preferable to use a different task version targeting a readily available execution unit type.

Should our scheduler not be able to determine a matching version where all hardware resources are available, and if the current task has a higher priority than the one currently using the targeted resource, we apply a Priority Inheritance Protocol (PIP) [29] and reschedule the task.

Going further with versions, we provide multiple configuration options to automatically select the version for the current job. At the time of writing it is possible to configure the version selection depending on (1) the current energy capacity (battery status) of the platform, (2) depending on an energy/time trade-off, (3) depending on the current execution mode[2], (4) depending on a bit mask permission, or (5) with a call to a user-defined function. The method to use is specified in the configuration header file, thus, only one method is effectively used at runtime, but switching is possible at compile time.

Each of these selection options requires different information from the user. This information is provided when declaring a version using *version_decl* through the *VSelect props* argument. The type of this argument is a structure morphed to cope with the selected method. For example, if the method to select the version is based on energy then the structure includes two fields to provide the energy budget of the task, and a user function to request the platform-dependent battery status. We provide an example in Section 3.6.

Limitation: Practically, task versions targeting a specific hardware accelerator start on a CPU core before they move the main workload to the accelerator, and eventually complete their execution back on a CPU core. For the time being, we consider the accelerator to be busy from the beginning of the initial CPU part to the end of the final CPU part. In the near future we plan to add an asynchronous mechanism, where CPU cores can be used by tasks while the accelerator-bound task actually runs on the accelerator. For initial work in this direction we refer the interested reader to [31].

3.3 Partitioned & Global On-line Scheduling

We rely on the concept of shielded processors, as described in [6, 33]. The idea is to reserve cores to only execute real-time (RT) tasks in order to minimise interference with system tasks. On each of the reserved cores we spawn one thread, called *worker thread* or *virtual CPU*. This thread serves as a container for the execution of the user real-time tasks.

An on-line scheduler must activate tasks following their arrival time (period), decide which version of the task to execute and dispatch tasks to a worker thread. YASMIN supports two modes: (1) *global*: all tasks can be executed on any virtual CPU; (2) *partitioned*: all tasks have a predefined target virtual CPU. The selection between the two modes is done at compile time through the configuration header file. Hence, only one of the two options is effectively compiled into the resulting binary. Switching between global and partitioned scheduling requires the modification of a single macro definition and a recompilation.

YASMIN supports static and dynamic priority assignments following task periods (rate monotonic), deadlines (deadline monotonic, earliest deadline first) or any statically user-defined priorities.

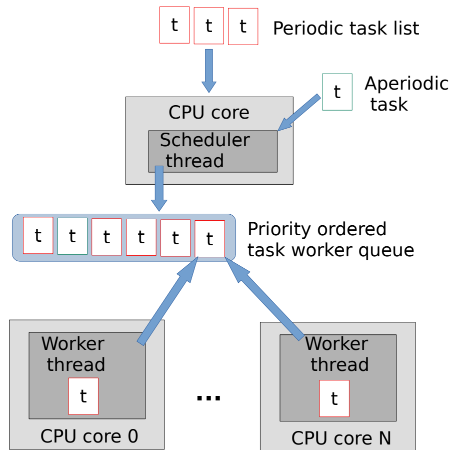
Specifically with DAG-based tasks, only the source nodes need to have a period attached. Intermediate and sink nodes are automatically activated by the scheduler once all required incoming data are present in their input channels.

Figures 1a and 1b illustrate our overall architecture for global and partitioned scheduling strategies, respectively. In both modes each worker thread is pinned to a specific core. With global scheduling all worker threads share a common ready queue, whereas with partitioned scheduling each worker thread has its own ready queue.

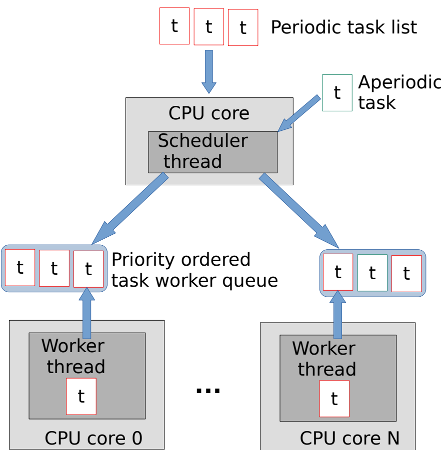
In either case, global or partitioned, the ready queue is filled by a separate scheduler thread that is likewise pinned to its private core. Unlike Saranya and Hansdah [33], who also use an external scheduler thread, we do not constantly check for new tasks to activate. Instead, we only periodically check for new tasks to schedule, i.e. between two activations the scheduler thread *waits*. The period of the scheduler thread is determined using the greatest common divisor of all declared task periods.

Using a separate scheduler thread that executes on its private core decreases parallelism: one core less is available to execute user RT tasks, but it increases predictability by minimising interference. For example with Linux, a core-pinned scheduler task of the kernel periodically preempts the running thread to check for higher priority tasks to execute. To meet real-time requirements this blocking

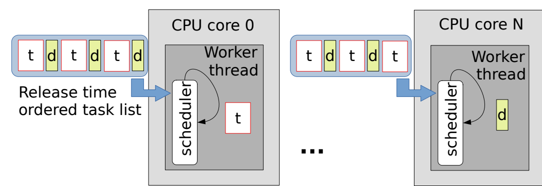
[2]For example, multi-security mode where different implementations of an encryption algorithm can be switched at runtime by changing the mode of execution.



(a) Global on-line scheduling strategy. The ready task queue is shared among worker thread.



(b) Partitioned on-line scheduling strategy. A scheduler thread pinned to another core feeds each worker thread ready task queue.



(c) On-line dispatcher with an off-line scheduling strategy. Each worker thread is pinned to a core, and a scheduling loop iterates on its ready task queue.

Figure 1: Illustration of the three scheduling classes supported by YASMIN: (a) global online scheduling, (b) partitioned online scheduling and (c) online dispatch with an offline schedule.

mechanism must be accounted for in the worst-case response time of user RT tasks. In practice, however, it is very difficult to estimate this blocking time spent in the kernel scheduler. Using a separate scheduler thread to check for higher priority tasks avoids such blocking and still allows preemption.

In addition, it is possible to configure the Linux kernel to prevent the aforementioned periodic scheduler task: a value of -1 needs to be written in the virtual file */proc/sys/kernel/sched_rt_runtime_us*. We refer readers interested in how to increase user control over time in Linux to [32].

Limitation: We do not support job migration. A job (task instance) spawned on a virtual CPU cannot be migrated to another one. However, we do support task migration: job i of some task may run on one virtual CPU while job $i+1$ runs on a different virtual CPU.

3.4 Off-line Scheduling

Unlike any similar middleware we found in literature, YASMIN natively supports *offline* computed schedules. An *off-line* schedule is computed before executing the application using the timing properties of the task set. In our runtime implementation an *online* dispatcher dispatches tasks at the predefined time following a given time table and a given mapping.

Figure 1c presents the overall architecture for the off-line scheduling strategy. Each worker thread is pinned to a specific core and has access to a predefined sequence of RT tasks ordered by increasing release times. Upon creation each worker thread starts executing a control loop running the RT task in order. To respect the release time of each task (computed off-line), special delay slots are added in between RT tasks that make the worker threads *wait* for a pre-determined duration.

If the static scheduler is aware of multi-version tasks, the version can be pre-selected off-line. This has the advantage of reducing the size of the resulting binary as it only needs to embed the actually required task versions instead of all task versions available to be potentially selected at runtime.

Limitation: We consider heterogeneous resource management to be handled by the off-line scheduling step. A task can, hence, target an accelerator without requesting access to the on-line dispatcher.

3.5 Further Implementation Aspects

This section describes other design issues we encountered and how we addressed them in YASMIN.

Accessing time: We access time using the POSIX primitive *clock_gettime* where the given clock can be set using the configuration file. As default, we employ *CLOCK_MONOTONIC* that gives a monotonically increasing clock with nanoseconds precision. The POSIX standard does not specify what the time 0 means. In Linux time 0 corresponds to system boot time. Our library stores the time at which the schedule is started using API call *start*. Afterwards, all timing information is computed using this initial starting time.

Pre-emption: YASMIN supports pre-emption with on-line scheduling policies only. Upon sorting, similar to [26], the scheduler thread sends a signal (PREEMPTION_SIGNAL), using the *pthread_kill* POSIX primitive, to each worker thread executing tasks with a lower priority than that of the head of the ready queue. This signal is caught by the thread which looks in the ready queue

for a higher priority task. If a higher priority task is found, a context-switch is operated. Upon completion, the process of finding a higher priority task is repeated until the initial preempted task becomes the task with the highest priority task and the context is switched back to it.

Context switching: Similar to [26] we use an architecture-dependent *swapcontext* function (in assembly code), which is called when switching execution context upon pre-emption. We draw inspiration from the GLibC swapcontext implementation, but leave out extra syscalls. As of writing, our swapcontext implementation is available for ARM 32/64 bits as well as X86-64 architectures.

Locking: Internally we implement synchronisation primitives, i.e. mutex locks and barriers, in two different manners: A first implementation uses the POSIX API implemented in the kernel and GLibC. A second implementation relies on lock-free algorithms from [25]. Users choose among the two options at compile time using the configuration file. We believe that lock-free algorithms form a superior choice for static WCET analysis [25], but spinlocks exhibit higher energy consumption. However, it is hard to analyse kernel and GLibC calls, but this solution offers better energy performance at the cost of predictability due to the kernel replacing the worker thread by an internal idle task. Selecting one or the other option depends on user preferences regarding predictability and energy conservation.

Waiting: With similar consideration in mind we provide the option to configure the waiting strategy in two ways: (1) *sleep* (default): calls some kernel code, which is hardly timing-analysable, (2) *spinlock*: enable a more precise overhead analysis at the cost of potential energy waste.

Protecting against page fault: Similar to [26] we lock our library code in memory using the POSIX primitive *mlockall*. This prevents swapping out the code of our library.

Interrupts: We set the kernel to use *threadirq*, and we shield the processor using *isolcpu*. Hardware interrupt handlers are composed of two parts, a top and a bottom part. We cannot do much about the top part that usually is pinned to a specific core. For the bottom part if they are not pinned to a specific core then the same configuration as for software interrupts applies. If they are specific to a core and this core runs a worker thread or the scheduler thread then their scheduling is left to the underlying OS. Care must therefore be taken to ensure that the priority of our worker threads and/or scheduler threads allows these bottom part interrupt handlers to execute.

3.6 Example

The following two listings 1 and 2 show an example of four tasks. The four tasks represent a fork-join graph where a fork is connected to two other tasks before joining to a join task. Data are exchanged using FIFO channels. The task *left* has two versions, one using a specific hardware accelerator and the other running on a CPU core. YASMIN is configured to select the version according to the current energy capacity of the platform.

Listing 1: Essential configuration example, must be in a *config.h* file

```
1  #include "yasmin_constants.h"
2  /*there 1 periodic task: the fork task*/
3  #define PERIODIC_TASK_SIZE 1
4  /*all other tasks are activated depending on
       the presence of input data*/
5  #define NONPERIODIC_TASK_SIZE 3
6  /*there are 4 FIFO channels connecting tasks*/
7  #define CHANNEL_SIZE 4
8  /*At most 2 versions are used*/
9  #define VERSION_MAX_SIZE 2
10 /*adapt the structure and code to select
       versions of task based on remaining energy
       .*/
11 #define VERSION_SELECTION ENERGY
12 /*One hardware accelerator is used*/
13 #define HWACCEL_SIZE 1
14 /*the example uses a global on-line scheduler
       */
15 #define MAPPING_SCHEME  GLOBAL
16 /*priority are given using EDF*/
17 #define PRIORITY_ASSIGNMENT EDF
18 /*2 worker threads will be used*/
19 #define THREADS_SIZE 2
```

Listing 2: C code example using YASMIN common API with user-defined priority

```
1  struct token { int value ; }
2  /*declare a dependency without data exchange*/
3  channel_decl(fl, char, 0);
4  /*declare dependencies with data exchange*/
5  channel_decl(fr, struct token, 1);
6  channel_decl(rj, int, 2);
7  channel_decl(lj, int, 1);
8
9  void fork(void *arg) {
10   struct token; token.value = 2;
11   channel_push(fr,token)
12 }
13 void right(void *arg) {
14   struct rec_token;
15   channel_pop(fr, &rec_token);
16   channel_push(rj, rec_token.value);
17   channel_push(rj, rec_token.value*2);
18 }
19 void join(void *arg) {
20    int rec_data;
21    channel_pop(rj, &rec_data);
22    channel_pop(rj, &rec_data);
23    channel_pop(lj, &rec_data);
24 }
25 void left_v1(void *arg) {
26   int *a = (int*) arg;
27   channel_push(l, *a);
28 }
29 void left_v2(void *unsued) {
30   int val = get_val_from_specific_accel();
31   channel_push(l, val);
32 }
```

```
33 /*User defined function to get the battery
       status*/
34 static void current_battery_level() { return
       .... ; }
35 void main(int argc, char **argv) {
36   TData f, j, r, l;        TID fid, jid, rid,
        lid;
37   VID lv1id, lv2id;     HID aid;
38   /*Due to the given configuration, the
        required information to select a version
         is energy budget*/
39   VSelect lv1_select, lv2_select;
40
41   init(); // initialise YASMIN
42
43   f.name = "fork"; f.period = 250;
44   //initialise other tasks
45   l.name = "left";
46   lv1_select.energy_budget = 5;
47   lv2_select.energy_budget = 12;
48   lv1_select.get_battery_status =
49     lv2_select.get_battery_status =
         current_battery_level;
50
51   fid = task_decl(&f, fork, NULL);
52   //declare other tasks
53   lid = task_decl(&l);
54   lv1id = version_decl(lid, left_v1,
        lv1_select);
55   lv2id = version_decl(lid, left_v2,
        lv2_select);
56
57   aid = hwaccel_decl("
        quantum_rand_num_generator");
58   hwaccel_use(lid, lv2id, aid);
59
60   channel_connect(fid, rid, fr);
61   channel_connect(rid, jid, rj);
62   //declare other channel connections
63
64   start(); // Start the schedule
65   //wait for some event
66   stop(); // Stop the schedule
67   cleanup(); // Cleanup before exiting
68   return 0;
69 }
```

4 EVALUATION

We empirically evaluate the overhead and latency introduced by YASMIN against various state-of-the-art task management alternatives. We target the embedded heterogeneous COTS platform Odroid-XU4 [20] as it provides multiple heterogeneous cores and allows us to run a RTOS with the support of the PREEMPT_RT patch set for Linux. The Odroid-XU4 platform includes an ARM big.LITTLE octa-core CPU and a Mali GPU. The CPU is split into two clusters: the LITTLE cluster contains four energy-efficient but computationally less powerful ARM Cortex-A9 cores while the big cluster embeds four computationally powerful but energy-greedier ARM Cortex-A57 cores. We configure the OS following our guideline to tame Linux and minimise interference between OS and application code [32]. All code is compiled with GCC 4.9 without optimisation ($-O0$), as is common to perform WCET analysis for real-time systems [40].

4.1 Comparison with Mollison and Anderson [26]

Mollison and Anderson [26] provide a library which performs a global earliest deadline first (G-EDF) schedule on behalf of the OS. The library spawns worker threads on cores similar to our approach, but it does not reserve one core for a scheduling thread as we do. Instead, they rely on a global queue, which is shared among all worker threads, and on test-and-set primitives to ensure mutually exclusive access to the global queue. The code provided by Mollison and Anderson only includes an x86 version. Therefore, we adapt the architecture-dependent part of the code to run this experiment on our ARM-based platform. We also adapt their method to measure time to match ours, thus ensuring a fair comparison.

Since Mollison's and Anderson's library targets homogeneous multi-core architecture, we successively use two and three big cores to execute user RT tasks. As YASMIN runs a separate scheduler thread, we map this thread on the remaining big core.

We use the task set generator based on the Dirichlet-Rescale (DRS) algorithm [18], which allows us to uniformly generate task sets with varying utilisation. We vary the number of tasks in the range $[20; 120]$. For each number of cores and for each number of tasks we generate 5 task sets, with utilisation varying in the range $[0.2; 2]$. This results in 1360 different task sets. The code related to each task is the same one as in [26], which is a simple function that iterates to reach a pre-defined WCET.

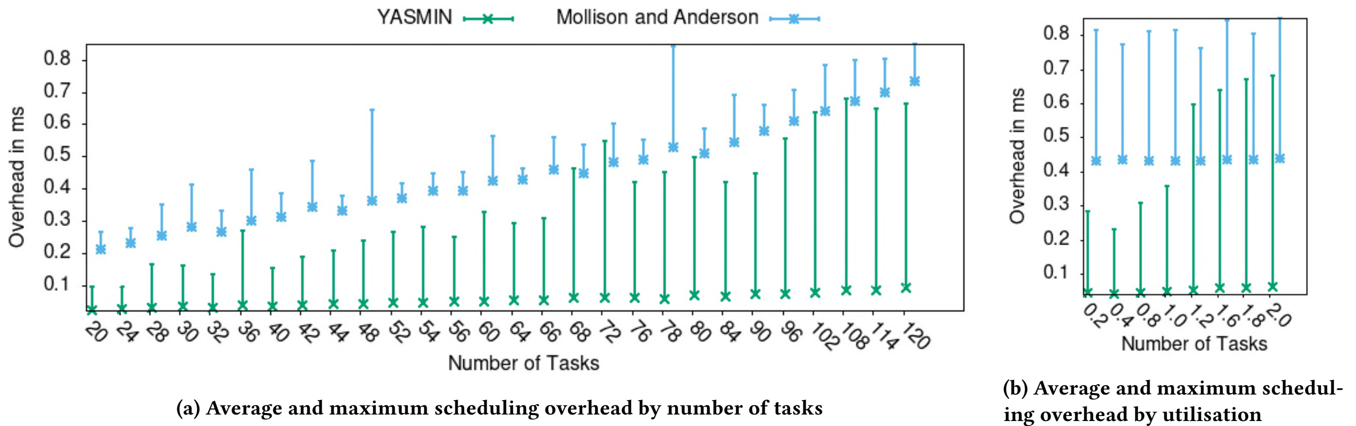
Figure 2 shows the evolution of the overhead depending on the amount of tasks and the total utilisation. On average YASMIN shows less overhead and better scalability in the number of tasks. However, the worst-case overhead observed with our library is a bit too high compared to the average, indicating one direction of future work.

4.2 Latency estimation comparison

Cyclictest[3] is a popular program used to accurately and repeatedly measures the response latency of a sporadic task activation. This program is available for *Linux+PREEMPT_RT* patch *Linux+SCHED_DEADLINE* and *Litmusˆ RT*. We adapt *cyclictest* to run under YASMIN management.

On the Odroid-XU4 board we switch between the different kernels, *Linux+PREEMPT_RT* patch and *Litmusˆ RT*, to perform each run of *cyclictest*. Unfortunately, we cannot include the *SCHED_DEADLINE* scheduler class in our comparison as it is not available for our platform, which further illustrates the motivation for creating YASMIN. Similarly, *Litmusˆ RT* offers several scheduling policies. However, *cyclictest* fails to execute with some of them, hence they are not included in this experiment.

[3]https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start



(a) Average and maximum scheduling overhead by number of tasks

(b) Average and maximum scheduling overhead by utilisation

Figure 2: Comparison of the overhead between YASMIN and [26]

We invoke *cyclictest* with the same condition:
-t 6 -d 0 -i 10000 -m -l 10000, i.e. we want 6 threads woken up 10000 times at the same time with a 10ms period and with locked memory. We restrict ourselves to 6 threads as our middleware library needs a 7th thread for scheduling and we leave one core available to the OS, again to improve predictability.

To generate interfering load on the platform we use the tool *stress-ng*[4], which we configure to stress the scheduler and the computing cores. *Stress-ng* is invoked with the following parameters: *-C 8 -c 8 -T 8 -y 8*, which roughly means that 8 threads are spawned per stressor, i.e. cache trashing, computation, timer events, sched_yield calls. For more details see the *stress-ng* documentation.

Table 2 displays the latencies we observed in the different configurations. The first column shows the kernel type and version used to gather the measurements. The second column displays the version of cyclictest used: *YASMIN* stands for our adapted version using our library, *RTapps* stands for the common version shipped with the *PREEMPT_RT* patch set and *litmus+XX* stands for the version shipped by *LitmusˆRT* where *XX* is the OS set up scheduler. The third column of Table 2 shows the minimum, maximum and average latency observed across the 6 threads.

On the Linux kernel with *PREEMPT_RT* the observed latency using YASMIN is similar to the initial *cyclictest* version, though slightly higher, due to our library overhead. When running on *LitmusˆRT*, we observe a higher overhead of our library compared to other versions. However, the benefit of *LitmusˆRT* comes at the price of no support for complex COTS heterogeneous platforms.

5 USE-CASE: UNMANNED AERIAL VEHICLE

An industrial partner provided us with a use-case involving an unmanned aerial vehicle (UAV), or drone for short, performing object detection on images. The goal of the use-case is to detect life boats or similar objects on sea while autonomously patrolling some geographic area to call upon rescue services if needed. Combining real-time requirements, rapid prototyping, scheduling exploration and task implementation exploration are the leitmotiv brought by our industrial use-case for the creation of our middleware. As compared to previous approaches [26, 33], YASMIN enables various scheduling policies, task models and platforms with a simple API.

[4]https://wiki.ubuntu.com/Kernel/Reference/stress-ng

Table 2: Latency comparison between YASMIN, Linux+PREEMPT_RT and LitmusˆRT

OS	*Cyclictest* version	Latency in μs $< min, max, avg >$
Linux +PREEMPT_RT 4.14.134-rt63	YASMIN	90, 1481, 500
	RTapps	176, 1550, 463
LitmusˆRT 4.9.30-litmus	YASMIN	67, 318, 170
	RTapps	33, 222, 74
	litmus+GSN-EDF	35, 247, 84
	litmus+P-RES	988, 1206, 1027

The UAV under study is a fixed-wing drone. The application scenario is a Search & Rescue (SAR) mission where the drone flies above the sea and sends an alarm to a ground station when it detects life boats. Figure 3 provides a graphical sketch of the system. The drone embeds multiple computing platforms that can be split into three parts: flight control, image capture and mission-specific payload application.

Flight Control: To fly in total autonomy the drone uses a GPS-based autopilot open-source software stack, called PX4[5], that pilots the drone following a pre-loaded mission. It runs on a PixHawk 2 platform[4] (single-core Cortex M4F with 256 KB RAM).

Image Capture: To capture images an Elphel[6] board with a camera is mounted below the drone. The Elphel board runs GNU/Linux; captured frames are streamed using standard GStreamer[5] libraries. The configured GStreamer pipeline streams out the image via a HTTP server, which is accessible through an Ethernet port on the Elphel board. The use of GStreamer to deliver images at a fixed frame rate is a requirement for the implementation of the system.

Search & Rescue Payload Application: The SAR application runs on a Toradex Apalis TK1[7] Computer-on-Module hardware

[5]https://px4.io/ – https://pixhawk.org/
[6]https://www.elphel.com/ – https://gstreamer.freedesktop.org/
[7]https://developer.toradex.com/products/apalis-tk1 – https://ubuntu.com/

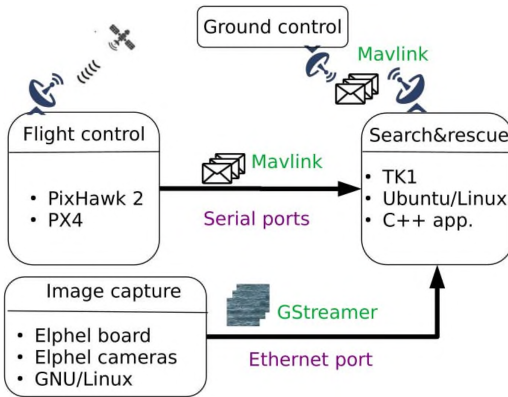


Figure 3: Overview of the SAR UAV system including hardware, operating environment and software

platform, which provides a quad-core ARM Cortex-A15 CPU, 2 GB of DDR3 RAM, and 16 GB of non-volatile storage. It also features an NVIDIA Kepler GPU with 192 cores. The GPU device can be exploited to accelerate image processing tasks. The board runs a modified Ubuntu/Linux[6], which includes NVIDIA proprietary drivers for the Kepler GPU. This precludes the use of both a Real-Time Operating System (RTOS) and the RT-patch set for Linux as neither of them supports this hardware platform.

The original SAR application code, as provided by our industrial partner, has mostly been developed in C++, with an object detection function written in CUDA. It receives Mavlink-encoded[8] messages from Flight Control through a serial port on the board. Among others these messages provide time synchronisation, update GPS coordinates, and enable/disable the payload application. The latter feature allows us to save energy by not running the SAR application while navigating to and from the mission area.

The SAR application also receives frames from Image Capture through its ethernet port. Upon reception of a *toggle image capture* message from Flight Control, a GStreamer pipeline is activated. It downloads a new frame at a fixed frame rate. This frame is stored in a queue until it is processed by the detection algorithm, which is likewise activated/deactivated by the same message. Due to the low speed of the drone there is no need for a high frame rate. The frame rate is set at 2 frames per second (fps). Upon detecting life boats a message is sent to Ground Control, including the number of boats, their corresponding GPS location and the image itself for manual validation.

Figure 4 shows a simplified view of the tasks within the SAR. There are two independent tasks, where one is a DAG with multiple nodes. The periodicity of each root node is presented in the figure along with their WCET. Furthermore, four tasks have multiple versions: Tasks *Detect objects*, *Highlight objects* and *Estimate speed* deal with images with either a CUDA-based accelerator implementation or a CPU-only implementation. The task *Encode* has two versions (or implementation variants) to either not encode the data (*Plain*) or use the AES algorithm for encryption (*AES*). The latter further

[8]https://mavlink.io/en/

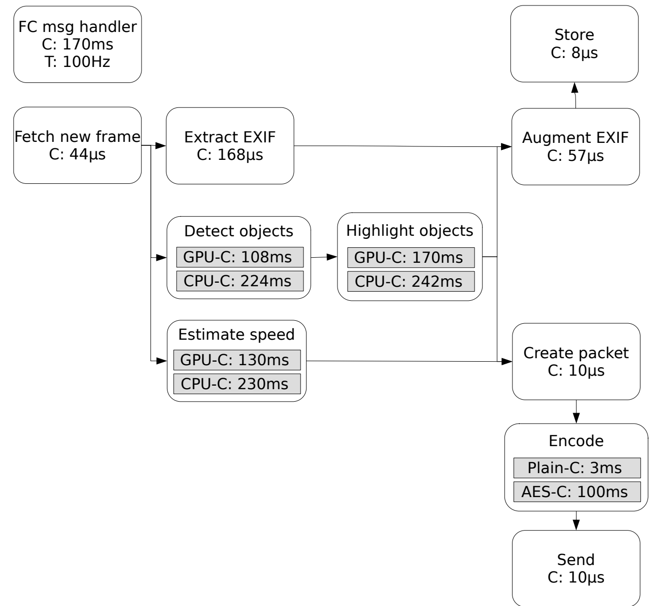


Figure 4: Simplified view of the SAR application tasks

allows two modes of execution: a normal mode and a secure mode. The secure mode is activated when boats are detected in the frame.

We crafted a simple mission to simulate an environment for the search-and-rescue application which we compile and link with our YASMIN middleware. Figure 5 shows the results of a wide range of experiments. In blue and using the left y-axis we show the minimum, average and maximum time observed to process a single frame. Remember that the search-and-rescue application demands a frame rate of 2 frames per second, which translates to a deadline of 500 ms. In red and using the right y-axis we show the deadline miss ratio: red crosses indicate the percentage of invocations that violate the 500 ms deadline. On the x-axis we play with a number of configurations. Firstly, we make use of four different scheduling policies from the YASMIN portfolio: global earliest deadline first (G-EDF), global fixed priority with deadline-monotonic priority assignments (G-DM), partitioned earliest deadline first (P-EDF) and partitioned fixed priority with deadline-monotonic priority assignments (P-DM). Next, we run each of the four scheduling policies with and without preemption. Finally, we run each of the above eight scheduling techniques with one of three possible configurations: force the scheduler to only use CPU versions of tasks, force the scheduler to only use GPU versions or permit the scheduler to choose between both alternatives for best performance.

As to be expected, the average time to process a frame is longer for CPU-only configurations, where we observe considerable numbers of deadline misses. The GPU-only configurations perform on average significantly better than the CPU-only configurations, but nonetheless exhibit deadline misses. The deadline miss ratio is even similar to the CPU-only configurations due to deadline misses of the stand-alone *flight control message handler* task. The only configurations that consistently avoid deadline misses are the ones

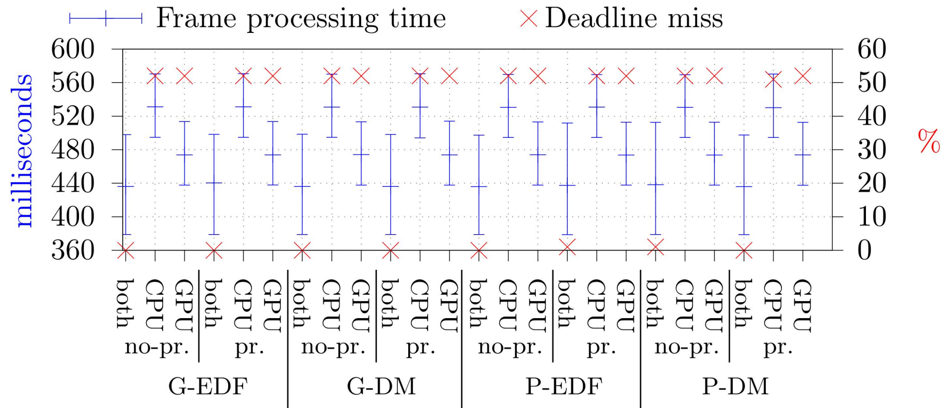


Figure 5: Scheduling exploration for the drone use-case

employing both CPU and GPU versions with automatic selection by the scheduler in YASMIN.

To summarise, all scheduling strategies (G-EDF, G-DM, P-EDF, P-DM) exhibit similar overhead and deadline miss ratios. Looking at the exact numbers, the partitioned strategies suffer from a slightly higher processing time per frame (6ms for the maximum values). This is due to the loss of the flexibility of partitioned scheduling versus global scheduling. This leads to a single deadline miss for P-EDF-both and P-DM-both for the *flight control message handler* task.

Our search-and-rescue unmanned aerial vehicle illustrates various benefits of our proposed YASMIN middleware:

- We can easily play with a variety of scheduling policies and priority assignment strategies to perform a systematic design space exploration to determine which strategy yields the best performance.
- YASMIN enables us to have multiple versions of tasks with different properties, e.g. running on the CPU or running on the GPU, and to select the best option dynamically at runtime. In our experiments this was the only option that produced zero deadline misses.
- YASMIN is portable and can run on top of any POSIX-compliant operating system.
- YASMIN can easily be extended by additional scheduling policies, etc, as it runs entirely in user space.

YASMIN can be used in two different ways: for manual systems engineering or as a compilation target from some higher level specification. As pointed out in the introduction, compiler-based synthesis is our goal in the overarching context. Either case grossly benefits from the ability to fundamentally manipulate the internal organisation of an application, independent of the application-specific code, by merely applying minor changes to some API calls. This way YASMIN enables systematic design space exploration across a range of scheduling policies and application configurations with minor effort in programming or compiler backend engineering.

6 RELATED WORK

Various forms of real-time operating systems (RTOS) [1, 7, 16], kernel patches [5] or hypervisors [24] have been proposed to control resources in the presence of real-time requirements. These solutions are known to enhance real-time scheduling capabilities, but they are to different degrees neither as portable nor as easy to maintain nor as customisable as a user-space library or middleware as YASMIN. They also hardly support heterogeneous platforms and are mostly limited to micro-controllers.

Mollison and Anderson [26] created a library to schedule a set of tasks in user space. Their library is intended to be used on top of a RTOS (Linux + PREEMPT_RT in their experiments). Target applications include sporadic task sets scheduled on multiple cores grouped in clusters (C-EDF). Their scheduling strategy supports dynamic priority (EDF), preemption and migration. Among scheduling capabilities and other issues, the library provides locking mechanisms with priority inversion (short wait time: spin-lock, long wait time: context switch), synchronisation protocols for critical sections as well as long system call handling mechanisms that avoid blocking the entire system. The authors also provide an empirical evaluation of the overhead induced by the library based on measurements.

Similar to Mollison and Anderson we abstract the scheduling capabilities of the kernel within worker threads, or virtual processors, each mapped (and pinned) to a specific core. These worker threads are responsible for the execution of the real-time user tasks and guarantee timing constraints. Unlike Mollison and Anderson, we do not allow job migration which makes our library simpler and more efficient at the expense of specific scheduling strategies. We advocate the reservation of a specific core for all non-RT tasks (mostly system tasks) and interrupt handlers, which allows us to provide the same guarantees in a simpler way using a COTS OS instead of a fully-fledged RTOS. Lastly, the implementation provided by Mollison and Anderson makes extensive use of dynamic memory allocation. This incurs well-known hazards for estimating the WCET of the library, thus losing confidence in the reported overhead.

ExSched [3] is a framework to allow scheduling from user space. It is composed of two parts: (1) a user space library providing a minimal API to an application program to set parameters and to control the beginning and end of a schedule; (2) a kernel space module which acts as a proxy between user-space API calls and kernel scheduling primitives. While the authors claim their method to be OS independent, the ExSched library requires a Linux kernel module to be loaded. This strongly links their user space API to the Linux kernel. However, nlike YASMIN, they do allow preemption and migration. An extension of ExSched to mixed-criticality workloads has been proposed in [19].

The ShedISA framework is introduced in [33] to enforce real-time constraints on COTS platforms. This framework comes as an extension of the Linux kernel by providing a new scheduling class called *SCHED_IS*. *SCHED_IS* extends *SCHED_DEADLINE* but comes with a higher priority within the kernel. It heavily uses processor shielding by splitting cores into three groups: *Linux cores* to execute system tasks, *Service cores* to execute the scheduler and *RT cores* to execute SchedISA RT tasks. They only support the P-EDF scheduling algorithm, but an extensive study of all induced overhead is presented.

SF3P (Scheduling Framework For Fast Prototyping) [14] is a framework to explore the design of a hierarchical composition of real-time schedulers. This type of scheduler can be represented as a tree of schedulers where the next task to schedule is decided by going from the root of the tree to a leaf. Walking through the tree following each stage scheduler decisions at the end will effectively schedule a workload on a core. The framework allows to quickly build this scheduler tree in order to test its viability. However, the proposed framework is not meant for deployment and does not provide any timing guarantees as its purpose is for design space exploration only.

Serra et al. [34] propose a complete middleware framework to enhance user experience regarding scheduling strategies from the Linux user space. From the user point of view, it facilitates the setup of existing scheduling strategies by hiding required invocations to syscalls (pthread_* API) that configure the environment. The framework is composed of a set of plugins, dynamically linked library, a daemon running in privileged mode (root) and a user library linked to the final application. Each plugin corresponds to a specific scheduling policy that will interact with the kernel using the currently available kernel API. These plugins are loaded by the daemon to apply the user configuration on threads in order to achieve the desired schedule. While the overall structure of the framework is kernel version independent, the related plugins are not, which makes retro-compatibility and future maintenance complicated. Each task is considered to be one thread which is not feasible for very large systems. Only task addition overhead is presented, leaving other overheads unknown. For example, interference with the daemon thread immediately comes to mind.

Similarly, Chishiro [9] proposes a middleware that sets the priority of threads in user space to influence the scheduling decisions made by the kernel. This middleware, called RT-Seed, targets real-time trading systems with a parallel-extended imprecise computation task model executing a partitioned semi-fixed priority scheduling algorithm. The type of targeted systems is limited to homogeneous processors, and the task model is generally unsuitable for embedded systems with real-time requirements.

Singhal et al. [35] propose to add a module to the kernel to add a scheduling level. This module will receive tasks (here tasks are processes) from the user-space and performs a schedulability test which, upon success, will compute the scheduling parameters. The upper scheduling level offered by the system will then schedule the processes according to their parameters. In order to achieve the desired scheduling policy, the module keeps track on which tasks have been added with which parameters to update them if necessary. A module is quite dependent of the current kernel version, at least to the major revision. No overhead analysis is presented in the paper, however.

Slite [12] supports control of the scheduling of an entire system at user level. It augments the Composite OS [28] with a direct mapping between kernel and user threads. Both levels exchange messages to maintain the coherence of active threads. This allows to account for interrupt threads placed by the kernel in the scheduling policy. They support partitioned (non-)preemptive fixed priorities and EDF where each core has it own scheduling thread.

Finally, Bristot de Oliveira et al. [10] show how to account for scheduling overhead within the Linux kernel. This helps in identifying where overhead occurs within schedule deployment.

There are some major differences between our work and the various frameworks and libraries mentioned before. Their common focus is on online scheduling strategies, whereas we additionally support pre-computed off-line schedules. On the task model they merely support independent tasks, whereas we work with DAG-based task models with dependencies. We are not aware of any previous work putting heterogeneous architectures into the focus. Neither are we aware of any previous work supporting multiple versions of tasks with automatic runtime selection.

7 CONCLUSIONS

We have presented YASMIN, a real-time middleware that performs the scheduling of an end-user application on behalf of the OS in user space. YASMIN significantly simplifies design space exploration with respect to scheduling and mapping by disentangling scheduling and mapping decisions from functional application code.

YASMIN is, to the best of our knowledge, the first middleware that embraces heterogeneity on embedded COTS platforms, among others through support for multi-version tasks. The major features of YASMIN include the possibility to easily switch between scheduling policies and to deploy applications using scheduling policies that are not available at OS level. All this can be achieved without the need to adapt the OS kernel or the end user application. We show that the overhead and latency induced by YASMIN is similar to existing state-of-the-art task management systems. We also show the applicability and benefit of using YASMIN and multi-version tasks on an industrial use case involving unmanned aerial vehicles in a search & rescue application.

YASMIN is a corner stone in using COTS hardware and operating systems for non-safety-critical real-time applications. This is crucial for modern heterogeneous embedded platforms that typically come

with specific OS versions/patches or proprietary drivers that preclude the effective use of specialised real-time OS support. YASMIN provides best possible guarantees on timing behaviour entirely in user space. The (non-trivial) taming of timing behaviour is hidden within the YASMIN implementation and provided to users *for free*. Still, we deliberately restrict ourselves to target non-safety-critical real-time applications as only specialised hardware and operating systems can provide formal timing guarantees.

As future work we plan to improve the management of real-time tasks with arbitrary activation patterns by using recurring servers, e.g. [13]. This would increase our support for real-time applications. We also plan to improve the support for heterogeneous platforms by adding a mechanism to provide asynchronous usage of hardware accelerators. Beyond GPGPUs we aim at supporting FPGAs, similar to Gracioli et al. [17].

ACKNOWLEDGMENTS

This work is supported and partly funded by the European Union Horizon-2020 research and innovation programme under grant agreement No. 779882 (TeamPlay project) and under grant agreement No. 871259 (ADMORPH project). This work is supported by COST Action CA19135 CERCIRAS funded by COST (European Cooperation in Science and Technology).

REFERENCES

[1] 2018. Erika Enterprise http://erika.tuxfamily.org/drupal.

[2] 2021. YASMIN: Yet Another Scheduler MIddleware for heterogeNeous COTS platforms. https://bitbucket.org/uva-sne/coordinationruntime

[3] Mikael AAsberg, Thomas Nolte, Shinpei Kato, and Ragunathan Rajkumar. 2012. Exsched: An External CPU Scheduler Framework for Real-Time Systems.

[4] Benny Akesson, Mitra Nasri, Geoffrey Nelissen, Sebastian Altmeyer, and Robert Ian Davis. 2020. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (Proceedings)*.

[5] Alessio Balsini. 2014. Adaptive Scheduling Parameters Manager for SCHED_DEADLINE. In *Workshop on Real-Time Scheduling in the Linux Kernel*.

[6] Steve Brosky and Steve Rotolo. 2003. Shielded processors: Guaranteeing sub-millisecond response in standard Linux. In *International Parallel and Distributed Processing Symposium (IPDPS)*.

[7] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. 2006. Litmusˆrt: A testbed for empirically comparing real-time multiprocessor schedulers. In *Real-Time Systems Symposium (RTSS)*.

[8] Davide Calvaresi, Mauro Marinoni, Arnon Sturm, Michael Schumacher, and Giorgio Buttazzo. 2017. The challenge of real-time multi-agent systems for enabling IoT and CPS. In *International Conference on Web Intelligence (WI)*.

[9] Hiroyuki Chishiro. 2016. Rt-seed: Real-time middleware for semi-fixed-priority scheduling. In *International Symposium on Real-Time Distributed Computing (ISORC)*.

[10] Daniel Bristot de Oliveira, Daniel Casini, Rômulo Silva de Oliveira, and Tommaso Cucinotta. 2020. Demystifying the Real-Time Linux Scheduling Latency. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[11] Christian Ferdinand and Reinhold Heckmann. 2004. aiT: Worst-case execution time prediction by static program analysis. In *Building the Information Society*. Springer.

[12] Phani Kishore Gadepalli, Runyu Pan, and Gabriel Parmer. 2020. Slite: OS Support for Near Zero-Cost, Configurable Scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[13] Teguh M Ghazalie and Theodore P. Baker. 1995. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems* (1995).

[14] Andres Gomez, Lars Schor, Pratyush Kumar, and Lothar Thiele. 2014. Sf3p: A framework to explore and prototype hierarchical compositions of real-time schedulers. In *International Symposium on Rapid System Prototyping (RSP)*.

[15] Joël Goossens, Xavier Poczekajlo, Antonio Paolillo, and Paul Rodriguez. 2019. ACCEPTOR: a model and a protocol for real-time multi-mode applications on reconfigurable heterogeneous platforms. In *Proceedings of the 27th International Conference on Real-Time Networks and Systems*.

[16] Giovani Gracioli, Antônio Augusto Fröhlich, Rodolfo Pellizzoni, and Sebastian Fischmeister. 2013. Implementation and evaluation of global and partitioned scheduling in a real-time OS. *Real-Time Systems* (2013).

[17] Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. 2019. Designing mixed criticality applications on modern heterogeneous mpsoc platforms. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[18] David Jack Griffin, Iain John Bate, and Robert Ian Davis. 2020. Generating Utilization Vectors for the Systematic Evaluation of Schedulability Tests. In *2020 IEEE Real-Time Systems Symposium (proceedings)*. York.

[19] Tarun Gupta, Erik J Luit, Martijn MHPVD Heuvel, and Reinder J Bril. 2017. Extending ExSched with Mixed Criticality Support — An Experience Report. In *International Conference on Software Architecture Workshops (ICSAW)*.

[20] HardKernel. 2017. *Odroid-XU4, User ManualReal-Time Systems Symposium.* https://magazine.odroid.com/odroid-xu4/

[21] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. 2017. The Heptane Static Worst-Case Execution Time Estimation Tool. In *Workshop on Worst-Case Execution Time Analysis (WCET)*.

[22] Zahaf Houssam-Eddine, Nicola Capodieci, Roberto Cavicchioli, Giuseppe Lipari, and Marko Bertogna. 2020. The HPC-DAG Task Model for Heterogeneous Real-Time Systems. *IEEE Trans. Comput.* (2020).

[23] Edward Ashford Lee and David G Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.* (1987).

[24] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. 2020. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*.

[25] John M Mellor-Crummey and Michael L Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)* (1991).

[26] Malcolm S Mollison and James H Anderson. 2013. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[27] Nvidia. [n.d.]. *Jetson TX2, presentation.* https://developer.nvidia.com/embedded/jetson-tx2

[28] Gabriel Parmer and Richard West. 2008. Predictable interrupt management and scheduling in the Composite component-based system. In *Real-Time Systems Symposium*.

[29] Ragunathan Rajkumar. 2012. *Synchronization in real-time systems: a priority inheritance approach*. Vol. 151. Springer Science & Business Media.

[30] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. 2020. Towards Energy-, Time-and Security-Aware Multi-core Coordination. In *International Conference on Coordination Languages and Models*. Springer.

[31] Julius Roeder, Benjamin Rouxel, and Clemens Grelck. 2021. Scheduling DAGs of Multi-version Multi-phase Tasks on Heterogeneous Real-time Systems. In *14th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2021), Singapore*. IEEE.

[32] Benjamin Rouxel, Ulrik Pagh Schultz, Benny Akesson, Jesper Holst, Ole Jørgensen, and Clemens Grelck. 2020. PReGO: a generative methodology for satisfying real-time requirements on COTS-based systems: definition and experience report. In *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. ACM.

[33] N Saranya and RC Hansdah. 2014. An implementation of partitioned scheduling scheme for hard real-time tasks in multicore linux with fair share for linux tasks. In *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*.

[34] Gabriele Serra, Gabriele Ara, Pietro Fara, and Tommaso Cucinotta. 2020. An Architecture for Declarative Real-Time Scheduling on Linux. In *International Symposium on Real-Time Distributed Computing (ISORC)*.

[35] Purnima Singhal, Amit Kumar, Upendra Ghintala, and Kunal Chakma. 2014. Extended Level Real time Scheduling Framework: Using a generalized non-real time platform. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*.

[36] Gimpel Software. [n.d.]. *PC-Lint Plus.* https://www.gimpel.com/pclp.html

[37] Z. Tian, W. Shi, Y. Wang, C. Zhu, X. Du, S. Su, Y. Sun, and N. Guizani. 2019. Real-Time Lateral Movement Detection Based on Evidence Reasoning Network for Edge Computing Environment. *IEEE Transactions on Industrial Informatics* (2019).

[38] Toradex. [n.d.]. *Apalis TK1, presentation.* https://www.toradex.com/computer-on-modules/apalis-arm-family/nvidia-tegra-k1

[39] Ovidiu Vermesan, Peter Friess, Patrick Guillemin, Sergio Gusmeroli, Harald Sundmaeker, Alessandro Bassi, Ignacio Soler Jubert, Margaretha Mazura, Mark Harrison, Markus Eisenhauer, et al. 2011. Internet of Things Strategic Research Roadmap. *Internet of Things – Global Technological and Societal Trends* (2011).

[40] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* (2008).