PReGO: a Generative Methodology for Satisfying Real-Time Requirements on COTS-based Systems*

— Definition and Experience Report —

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

Ulrik Pagh Schultz
University of Southern Denmark
Denmark
ups@mmmi.sdu.dk

Benny Akesson
University of Amsterdam
TNO
Netherlands
k.b.akesson@uva.nl

Jesper Holst
Sky-Watch A/S
Denmark
jsh@sky-watch.com

Ole Jorgensen
Sky-Watch A/S
Denmark
olj@sky-watch.com

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

Abstract

Satisfying real-time requirements in cyber-physical systems is challenging as timing behaviour depends on the application software, the embedded hardware, as well as the execution environment. This challenge is exacerbated as real-world, industrial systems often use unpredictable hardware and software libraries or operating systems with timing hazards and proprietary device drivers. All these issues limit or entirely prevent the application of established real-time analysis techniques.

In this paper we propose PReGO, a generative methodology for satisfying real-time requirements in industrial commercial-off-the-shelf (COTS) systems. We report on our experience in applying PReGO to a use-case: a Search & Rescue application running on a fixed-wing drone with COTS components, including an NVIDIA Jetson board and a stock Ubuntu/Linux. We empirically evaluate the impact of each integration step and demonstrate the effectiveness of our methodology in meeting real-time application requirements in terms of deadline misses and energy consumption.

ACM Reference Format:

Benjamin Rouxel, Ulrik Pagh Schultz, Benny Akesson, Jesper Holst, Ole Jorgensen, 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 (GPCE '20), November 16–17, 2020, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3425898.3426954

*This work is supported and partly funded by the European Union Horizon-2020 research and innovation programme under grant agreement No. 779882 (TeamPlay) and by the Netherlands Organisation for Applied Scientific Research TNO.

GPCE '20, November 16–17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8174-1/20/11…$15.00
https://doi.org/10.1145/3425898.3426954

1 Introduction

Applying real-time systems theory to industrial systems is challenging as such systems frequently fail to meet the underlying assumptions of models and analyses. While theories often make idealised assumptions, industrial systems tend to use unpredictable hardware and software libraries, proprietary device drivers, and operating systems with timing hazards. Moreover, software architectures are often not designed with real-time constraints in mind, neither are many software engineers trained in real-time concerns.

Addressing this challenge is essential for successful deployment in industrial scenarios. We present a generative, component-based methodology to pragmatically providing real-time guarantees for embedded systems under real-world constraints: Team**P**lay **Re**al-time **G**uarantees for C**O**TS-based Systems (PReGO). PReGO includes the identification of real-time properties in the software, guidelines for taming a Linux OS, engineering principles to use the TeamPlay[1] component-based architecture and declarative specification language [29], and methods for applying the generative steps performed by our tool. We provide a systematic approach that can be applied to a wide range of Commercial-Off-The-Shelf (COTS) platforms running COTS Operating Systems (OS).

We evaluate PReGO using a maritime Search & Rescue (SAR) application executing on a fixed-wing drone, manufactured by our industrial partner Sky-Watch[2]. The system embeds multiple computing boards, including a heterogeneous platform that executes the SAR application. The COTS

[1] www.teamplay-h2020.eu
[2] www.sky-watch.com

hardware and OS have been selected by our industrial partner, aiming for a low-power platform with sufficient image processing performance. The application software has been designed according to the common practice of optimising for average performance. The whole original system performs well under average operating conditions, but it is not guaranteed to satisfy its requirements in a worst-case scenario.

This paper addresses the problem of pragmatically satisfying real-time guarantees under worst-case operating conditions using a mostly generative method. PReGO is designed to be applicable to a wide range of embedded Linux-based systems and DAG-based component software [1]. The four main contributions of this paper are:

1. PReGO, a partially generative methodology to pragmatically adapt the system and derive the necessary timing parameters to enable the use of an existing schedulability analysis;
2. a generative component-based tool flow supporting our methodology that turns a declarative specification of the components, their dataflow dependencies, and their timing requirements into an application infrastructure for C-based real-time programs;
3. details on how to improve the timing predictability of the underlying stock Ubuntu/Linux environment;
4. a step-by-step quantitative evaluation of the worst-case performance, memory consumption, and energy consumption of the SAR application that demonstrates the impact and trade-offs of the proposed adaptations.

The evaluation shows that after system integration with PReGO the system does not miss any deadlines in the worst-case scenario. At the same time energy consumption is reduced by 18% compared with the original code, thus increasing the flight time of the drone.

The rest of this paper is organised as follows. First, we present the PReGO methodology in Section 2. Then, we describe the case study including hardware, operating environment, and software in Section 3. In Section 4 we apply our methodology to the case study. Finally, we review some related work in Section 5 and conclude in Section 6.

2 Methodology

This section introduces the PReGO methodology for adapting legacy software running on COTS Linux-based platforms to meet real-time requirements. We begin with an overview of the methodology and then present all details step-by-step.

2.1 Overview

Fig. 1 illustrates PReGO, distinguishing between manual and tool-supported, automatic steps. There are two main workflows: 1) adapting the legacy software and 2) configuring the COTS platform.

The main workflow starts with the *identification of the Worst-Case Execution Time (WCET)* scenario. It is followed

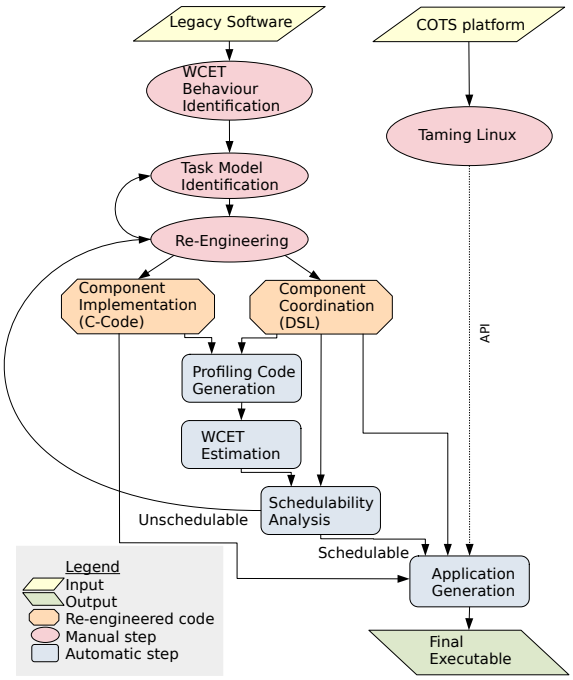


Figure 1. Overview of the PReGO methodology

by the *task model identification* that identifies the relevant system tasks. In the subsequent *re-engineering* step an existing implementation is (manually) refactored into identifiable *component implementations*.

At the heart of our generative approach, *component coordination* uses a declarative DSL that defines the external behaviour of the components and their orderly interaction [29]. The coordination DSL code describes the application architecture at a high level of abstraction. Throughout the paper we will use the terms *component* and *task* interchangeably. While the former is the common term in coordination programming, the latter is common terminology in the real-time domain.

From the coordination DSL code we automatically generate profiling code and obtain *WCET estimations* for individual component implementations. Subsequent *schedulability analysis* determines component priorities and mappings. If it turns out that our componentised application is not schedulable under the given hardware and time constraints, we must go back to the (manual) *re-engineering* step. Otherwise, we automatically generate the necessary application architecture code and link it with the separately compiled component implementations (now without profiling) into the *final executable* using platform configuration and API.

Independently, we configure (or *tame*) the COTS platform to provide time-wise more predictable execution properties, both for profiling component implementations and execution of the final application.

Not every application of PReGO starts with legacy software. When building new software from scratch the first

three steps of our methodology become obsolete, and we start with a componentised application architecture expressed in our component coordination DSL and the corresponding component implementations in C. In this case PReGO becomes fully automated and all steps are tool-supported.

2.2 Worst-case Behaviour Identification

Identifying the worst-case behaviour is not generalisable as it depends on the application and its operating environment. The idea is to find the worst-case conditions (e.g. input data and environment) for each task that will lead to the worst-case behavior. For instance, in an image processing algorithm the worst-case behaviour could be triggered by the largest possible image. We further elaborate on application-specific worst-case behaviour identification in Section 4.1 when applying PReGO to our SAR application.

2.3 Taming Stock Linux

The stock Linux kernel does not provide real-time capabilities. One could apply the patch set *PREEMPT-RT*, but in practice this is often not possible, e.g. in the presence of proprietary drivers. In the following we propose five customisations of the GNU/Linux environment that pragmatically reduce the interference between the system and application tasks without touching the kernel.

First, a major source of interference from the kernel are interrupt handlers. An interrupt is an event requesting immediate attention from the processor. The processor must then stop the currently executing code to run the interrupt handler. To limit the interference of interrupt handlers on our tasks, we configure the OS to map all interrupt handlers onto a specific CPU core on which no application tasks are scheduled. We achieve this by setting the selected core id in */proc/irq/default_smp_affinity* and all */proc/irq/*/smp_affinity*.

Second, the scheduler in the Linux kernel is designed to manage resources system-wide and per CPU. Hence, each CPU periodically executes a kernel scheduler task to examine its current state or perform some housekeeping. This feature is called *real-time scheduler throttling* and creates undesirable interference with application tasks. We disable this OS feature by writing the value −1 in */proc/sys/kernel/sched_rt_runtime_us*.

Third, another major source of interference from the OS are a set of services provided in a stock Ubuntu/Linux. While these services are convenient for general purpose usage, they may pollute the cache when scheduled on the same core. All extraneous services usually started at boot time, e.g. the printing service daemon *cupsd*, are disabled. Disabling these services decreases the number of processes/threads that potentially interfere with our tasks and reduces the memory footprint of the system.

Fourth, shared resources such as storage devices or memories can cause interference when accessed concurrently. Partitioning such devices avoids this interference by separating physical accesses. For example, if a shared storage devices is a bottleneck, adding an extra storage device and distributing accesses among them reduces interference and yields more control over temporal behaviour.

Fifth and last, to further increase control on the OS we use the POSIX primitive *pthread_setaffinity_np* to bind each thread to a single physical core. This way we effectively disable OS-level mapping. Depending on the hardware platform, however, the system CPU governor may conflict with this. The CPU governor controls the on/off state of all cores and continuously adapts the number of active cores to the workload. However, from a timing perspective the CPU governor adds thread migration cost that is difficult to assess. To increase our control of the OS and its predictability, we disable the CPU governor; the precise way to do so is platform-specific (detailed further in Section 4.2).

2.4 Task Model Identification

A key step towards providing real-time guarantees is to identify the system tasks and their associated timing parameters. A plethora of task models exist, all with different key properties. In the following we restrict ourselves to the ones supported by our methodology. First, tasks can either be dependent, modelled as a graph [36], or independent, e.g. [20, 24]. When dependent, the whole graph becomes the task, and timing properties are attached to the graph [3]. Second, the activation rate of each task in our context is a strict periodicity [20]. Third, a task model has a deadline constraint, which can be

- *constrained* when the deadline is less than or equal to the period,
- *implicit* [20] when the deadline is equal to the period, or
- *arbitrary* [24].

We support a Directed Acyclic Graph (DAG) task model [1], where tasks exchange data, also called tokens. A DAG task model can be identified when the predecessor/successor structure is not constrained (except the absence of cycles), consumption/production rates are equal for each edge, and each node produces and consumes the same amount of tokens at each instantiation. This includes some common task models that are related to the DAG model and are also supported by our approach: Synchronous DataFlow graph (SDF)[3] [18], where consumption and production rates may be different for an edge (the source node produces a different number of tokens than the sink node consumes at each instantiation) and the structure is not constrained; and Fork-Join Graph [38], which is an SDF where the structure is constrained, enforcing that only specific nodes can have

[3]SDF can be either cyclic or acyclic; we assume that if there is a cycle, initialisation tokens (known as delay tokens) are present on back edges, thus leaving the DAG theory applicable.

```
1  app Example {
2    datatypes {
3      (int, "int") (type_t, "struct complexType")
4    }
5    components {
6      TaskA { outports [(out, 42, int)]
7        deadline 10ms
8        period 1Hz
9      } // 42 int per second
10     TaskB { inports [(in, 42, int)]
11       outports [(out, 21, type_t)]
12     } // two ints become one type_t
13     TaskC { inports [(in, 21, type_t)] }
14   }
15   edges {
16     TaskA.out -> TaskB.in
17     TaskB.out -> TaskC.in
18   }
19 }
```

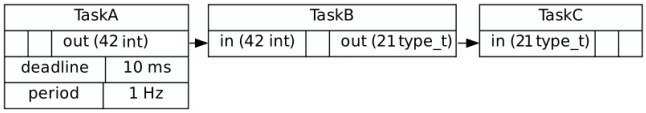


Figure 2. Example of a component coordination DSL specification of a simple pipeline of three successive components

multiple successors (split) or multiple predecessors (join), and enforcing that each split corresponds to a join. SDF and fork-join graphs need to be expanded in order to properly schedule them [18, 34]. This expansion step can be automatically done by our tool. Following that the resulting graph can be analysed using the more general DAG-oriented analyses.

Task Model Definition. We define an application as a group of tasks Γ, where each task $\tau_i \in \Gamma$ is defined as $\tau_i = (V_i, E_i, T_i)$ where V is the set of nodes (or subtask), E is the set of edges and T is the period. Then, within a task, each subtask $v_{i,j} \in V_i$ is defined as $v_{i,j} = (C_{i,j}, W_{i,j})$, where $C_{i,j}$ is the WCET estimate, and $W_{i,j}$ the Worst-Case Energy Consumption estimate (WCEC).

2.5 Task Re-engineering

Once the tasks have been identified, a re-engineering step is required to make the code reflect the task model and enable the generation feature available in our tool. The code body of each task must be extracted from the application and placed in separated functions. The core feature of the task should not include any form of code for inter-task communication, inter-task synchronisation, or task management. These mechanisms will later automatically be generated by our tool, see Sec. 2.8.

In addition to the code of each task, our tool requires a specification of the tasks and their interactions (the edges in a DAG-based task model). To this end, we reuse our coordination DSL, first described in [29]. We show an example program and its graphical illustration in Fig. 2. Each task is described with a set of input and output ports, which are connected to ports of other tasks. Each connection specifies the amount and type of data exchanged between tasks. From this DSL program our tool automatically checks stability of data production and consumption rates (deadlock) and type correctness, generates a special version of the application for profiling, performs a schedulability analysis, and generates the final version of the application. Automatically generating multiple artifacts from a single model in this manner is a best practice of model-based engineering and ensures that artifacts are always consistent with the model [33]. Model-driven engineering using tailor-made DSLs becomes increasingly common in industry [42], including the domain of embedded or cyber-physical systems [19]. The model-driven approach generally benefits development time and improves customisation, maintenance and evolution of software [2].

2.6 Worst-Case Execution Time

The WCET behaviour of each task can either be estimated through static code analysis or via profiling [43]. Vendors typically do not reveal detailed information about the timing behaviour of their COTS hardware. We are not aware of any static analysis tool that would be universally applicable to real-time Linux-based systems. Therefore, the only generally applicable method to estimate WCETs is to use measurements.

For the purpose of the WCET analysis, a sequential version of the application can be automatically generated, where each task is executed in sequence, respecting dependencies. The purpose of this approach is to remove task interference during WCET profiling, so-called WCET estimation in isolation [22]. We instrument each task call to record start and stop times. All tasks are executed on the same core, and the cache is flushed before each call to the task code, which triggers worst-case timing behaviour.

As tasks may exchange data, the communication time must also be included to guarantee the availability of data when consuming tasks are scheduled. Data exchanged between tasks within a graph is done using shared memory. Reading (resp. writing) the consumed (resp. produced) data by a task using shared memory involves loading (resp. unloading) the cache. We flush the cache prior to each call. Hence, the data transmission cost is included in our measured WCET.

Tasks may include computations across heterogeneous hardware, for example both a CPU part and a GPU part. In this case we employ a synchronous mechanism that stalls the CPU when the GPU is computing. Therefore, the measured WCET/WCEC for these particular tasks cover both parts of the task.

Our sequential version for the WCET analysis does not feature any parallelism. In the final parallel implementation tasks are implemented as threads, as it is the only execution container (with processes) available within our OS. Scheduling them implies context switching cost paid each time the executing thread on a core is changed. This cost is not

reflected in our measured WCET. Therefore, we must add an estimated, platform-specific context switch overhead to our measured WCET. Because estimations do not provide an upper bound of the actual WCET, we add an arbitrary safety margin of 20%, as in [30], to our estimated values to account for unknown or undocumented overheads in the COTS hardware and software.

2.7 Scheduling

Scheduling is usually performed by the Operating System (OS). When using COTS components, available choices are constrained by the OS supporting the platform. Depending on the available options, our tool automatically computes the schedule off-line, or it decides online. Online scheduling opens up more configuration opportunities regarding task-core mapping, priorities, etc. We here focus only on configuration supported by our tool, and on-line scheduling. As of writing, we support Partitioned (when task are mapped on a specific core), no Preemption, Fixed priorities and no Migration (hereafter referred as *PnPFnM*) scheduling policy. As advocated by Casini et al. [8], we, too, believe that this specific on-line scheduling policy is the rational choice for bringing maximum control on the timing on a COTS-based system not tailored for controlling timing behaviour. Partitioned scheduling gives us control over which task is executed on what core. Non-preemptive scheduling has the benefit of reducing interference among tasks executing on the same core. Lastly, fixed priorities allow us to influence the execution order of tasks sharing a core.

Upon selection of an online scheduling policy, validating a real-time system requires performing a schedulability analysis to *a-priori* guarantee that the system is schedulable. Should a system be deemed unschedulable, it is required to identify the root cause and to restart the re-engineering process. Because the WCET does not capture scheduling decisions, the difference between the start of a task and its completion can be larger than the WCET due to, among others, scheduling overhead. On the other hand, the Worst-Case Response Time (WCRT) captures this environmental blocking time in the worst condition. A schedulability analysis determines the WCRT for each task and checks whether it is less than or equal to its deadline. If this holds for all tasks, the task set is schedulable.

A schedulability analysis is strongly linked to both the task model and the scheduling policy. Hence, a vast range of literature addresses this research topic for every possible pair of task model and policy [15]. To deal with our PnPFnM scheduling policy, Casini et al. [8] derived a schedulability analysis that fits our DAG-based task model. Therefore, we implement their analysis in our tool to automatically check for schedulability.

2.7.1 Mapping Algorithm. Partitioned scheduling policies, such as our PnPFnM, require us to map tasks to cores prior to schedulability analysis (for online scheduling) or offline scheduling. Algorithms that map tasks to cores essentially solve the well-known bin-packing problem. Therefore, any algorithm addressing this problem is a potential candidate to determine how to map tasks to cores. More focused algorithms exist to perform this partitioning step, e.g., Wang et al. [41] use a machine learning approach. However, a simple space exploration, as presented by Casini et al. [8], is also possible when the number of tasks and cores are low.

2.7.2 Priority Assignment. Fixed priority scheduling policy, such as our PnPFnM, requires assigning a priority to each task prior to performing a schedulability analysis or to scheduling this application. According to Davis et al. [15], the Deadline Monotonic (DM) strategy is the most frequently used as it is simple to implement and yields optimal results in many cases. It assigns higher priority to tasks with shorter deadline.

2.7.3 Enforcing the Schedule. The targeted OS offers us different, limited real-time scheduling possibilities with three main schedulers in a stock Linux kernel: *SCHED_FIFO*, *SCHED_RR*, *SCHED_DEADLINE*. All of them queue threads that are ready to execute. The next scheduled thread is the one with the highest priority. *SCHED_FIFO* and *SCHED_RR* implement a fixed priority based scheduling policy, whereas *SCHED_DEADLINE* implements a dynamic priority scheme and, hence, can not be used with our aforementioned scheduling configuration.

Enforcing fixed priorities is straightforward. At the creation of a new thread using the *pthread_create* POSIX primitive, we can set the assigned priority for the corresponding task using the priority assignment already derived. Enforcing a partitioned scheduling policy is also straightforward using the POSIX primitive *pthread_attr_setaffinity_np* before the thread is created. The choice of mapping is performed using the mapping already derived. Finally, enforcing a non-preemptive schedule is realised using the aforementioned technique to limit the interference with the *real-time scheduler throttling* mechanism, as described in Sec. 2.3.

2.8 Generating the Final Application

The final step of our methodology is the automatic generation of all synchronisation, communication and task management code. Fig. 3 shows the code our tool generates for *TaskB* from Fig. 2. We likewise generate initialisation code and the entire *main*-function, but space limitations prevent us from showing more code here.

2.8.1 Execution Container. The idea behind an executing container is to decide if a component or a group of components should be executed within a process or a thread. For example, a developer wishing for full isolation of components may prefer to execute each component in its own process. Each task can be executed in its own thread or process, and when partitioned, a group of tasks can be executed

```
1  typedef struct {
2    size_t head; size_t tail; int fifo[42];
3  } fifo_int42_t;
4  char pop_int42(fifo_int42_t *e, int* res) {
5      if(e->tail == e->head) return 0;
6      *res = (e->fifo)[e->tail];
7      e->tail = (e->tail+1)%42;
8      return 1;
9  }
10 /*type_t is used as the type in Figure 2*/
11 void push_type_t21(fifo_type_t42_t *e, int val) {
12     (e->fifo)[e->head] = val;
13     e->head = (e->head+1)%42;
14 }
15 fifo_int42_t TaskB_in;
16 fifo_type_t21_t *TaskB_out = &TaskC_in;
17 void * __TaskB_(void* unused) {
18     int status; int in[42]; type out[2];
19     for(size_t i = 0 ; ; ++i) {
20         status = sem_wait(TaskA_out_lock);
21         for(size_t j = 0 ; j < 42 ; ++j)
22             pop_int42(TaskB_in, &in);
23         TaskB(in, &out); // user code of TaskB
24         for(size_t j = 0 ; j < 21 ; ++j)
25             push_type_t21(&TaskB_out, out);
26         sem_post(TaskB_out_lock);
27     }
28     return NULL;
29 }
```

Figure 3. Code snippet generated for TaskB from Fig. 2

within the same thread/process. Lines 17-29 in Fig. 3 show the thread function for *TaskB*.

In order to improve fault tolerance, the TeamPlay component framework has been extended such that each task graph is a process and each subtask (node in the graph) is a thread. This separation has been implemented to allow an automatic relaunch of a crashing process: when a thread crashes, the whole process crashes, but a crashing process does not affect other processes.

2.8.2 Task Management. There are two types of events to release a task: the beginning of a new period and the completion of a predecessor. First, we implemented new period releasing using alarms available within Linux: one alarm per component. Second, we implemented a semaphore-based mechanism to manage predecessor and successor (Lines 20 and 26 in Fig. 3).

2.8.3 Intra-component Communication. A dependency in graph-based task models implies a data exchange between a source and a sink task. Such dependencies are materialised with a FIFO channel. To access this channel we use the primitives push and pop [38]. They push tokens produced by the source into the channel, and they pop tokens consumed by the sink, respectively. The types of tokens and their transmitted amount are application dependent. The code generator produces an implementation of the aforementioned primitives for each required type. These types are extracted from the task re-engineering produced file, see Sec. 2.5. FIFO channels can also be implemented using different techniques depending on the target OS. We generate buffers implemented with shared memory (Lines 1-16 in Fig. 3).

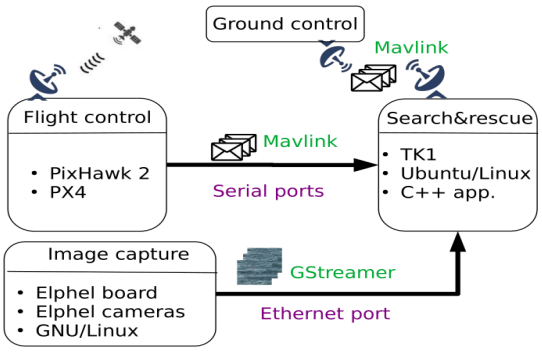
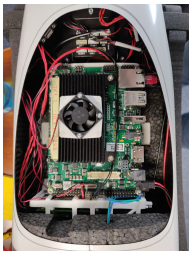


Figure 4. Overview of the system including hardware, operating environment, and software

(a) Bottom-side view with the two Elphel cameras



(b) Top-side view with the Apalis TK1 and the PixHawk

Figure 5. Drone installation

3 Case Study Overview

This section presents our industrial case study including hardware and operating environment. The system under study is a fixed-wing drone manufactured by our industrial partner Sky-Watch[4]. 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. Fig. 4 provides a graphical sketch of the system. The drone embeds multiple computing platforms that can be split in three parts: flight control, image capture, and mission-specific payload application (here SAR).

3.1 Flight Control

To fly in total autonomy the drone uses a GPS-based autopilot software stack that pilots the drone above the mission area loaded into the drone before taking off. The system uses an open-source autopilot software stack called PX4[5], which

[4]www.sky-watch.com
[5]https://px4.io/

runs on top of a PixHawk 2 platform[6] (single-core Cortex M4F with 256 KB RAM).

The PX4 software communicates with our payload application using Mavlink-encoded[7] messages sent through a serial port on the PixHawk 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.

3.2 Image Capture

To capture images an Elphel[8] board with two cameras is mounted below the drone. The two cameras have a $51°$ field of view and are mounted at a $30°$ angle forward, as shown in Fig. 5a. Only a single camera is used in this case study, the other is disabled to avoid interference. The Elphel board runs GNU/Linux; captured frames are streamed using standard GStreamer[9] 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.

3.3 Search & Rescue Payload Application

The SAR application runs on a Toradex Apalis TK1[10] Computer-on-Module hardware 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. Fig. 5b shows the TK1 board mounted inside the drone.

The board runs a modified Ubuntu/Linux[11], 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. This greatly increases the challenge of providing real-time guarantees.

The original SAR application code, as provided by our industrial partner, has mostly been developed in C++, with an object detection function in CUDA. The application is split into different processes and threads, thus enabling parallel execution. Processes communicate with each other using a socket mechanism. All processes and threads are, in this original version, scheduled using the default Linux scheduling policy (*SCHED_OTHER*, a completely fair scheduler).

The SAR application receives messages from Flight Control through a serial port on the board and frames from the Image Capture through its Ethernet port. Upon reception of a *toggle image capture* message from flight control, a GStreamer pipeline is activated, that downloads a new frame by accessing the HTTP server running on the Elphel board. This frame is stored in a queue until it is processed by the detection algorithm, which is likewise activated/deactivated by the same message.

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. A second GStreamer pipeline from the same camera records a video of the flight. It uses the same frame fetching method as above and is intended for post-mission verification.

Due to the low speed of the drone, there is no need for a high frame rate. Depending on the requested altitude, the frame rate can go from 2 to 10 frames per second (fps). The higher the altitude of the drone is, the wider area a single frame covers, and the lower the frame rate can be. To reduce the number of false positives, a restriction is given on the minimum size of detected objects. To be marked as life boats, detected objects must be wider that $0.5m^2$ on the picture. Obviously, the number of pixels needed for the size of the object depends on the parameters of the cameras (e.g. the focal) and the altitude.

4 Applying the PReGO Methodology

This section continues the demonstration of the PReGO methodology introduced in Section 2 by applying it step-by-step to the case study from Section 3.

4.1 Worst-case Behaviour Identification

The initial system works reasonably well under typical operating conditions. However, to ensure life boats are not left undetected, we need to provide real-time guarantees in a worst-case scenario (WCS), for which defining parameters is a challenge.

Examining the application software provided by our partner, we concluded that the WCS is triggered by the maximum number of detected objects in a frame when they are at their minimum size (determined by the altitude). While at the highest possible altitude (about 1000 m), it is theoretically possible to have over 5 million boats in a single frame, this worst WCS is unlikely to happen in real life. Together with our industrial partner, we decided to consider the life boat capacity of the biggest Oasis-class cruise ship as a reference to create a worst-case image with the maximum considered number of life boats, estimated to be 440.

To enforce the WCS we need to simulate a worst-case image stream that triggers the worst possible execution time for each frame processing operation. We run the SAR application under worst-case operating conditions given by our partner, which is 1000 m altitude, a frame rate of 2 fps, and a continuous stream of the worst-case image. Fig. 6 reports the execution time per frame in this WCS. We observe that there

[6] https://pixhawk.org/
[7] https://mavlink.io/en/
[8] https://www.elphel.com/
[9] https://gstreamer.freedesktop.org/
[10] https://developer.toradex.com/products/apalis-tk1
[11] https://ubuntu.com/

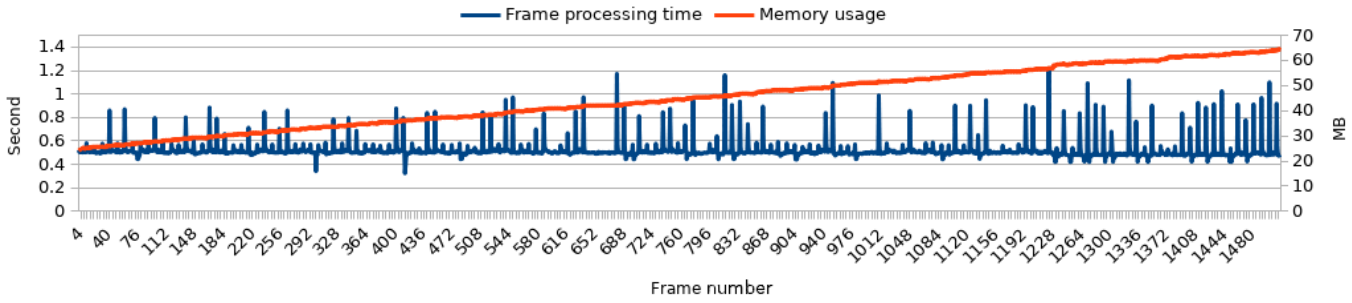


Figure 6. Original code running on a fresh installed Ubuntu/Linux in a WCS. AVG energy consumption: 3.8 J/frame.

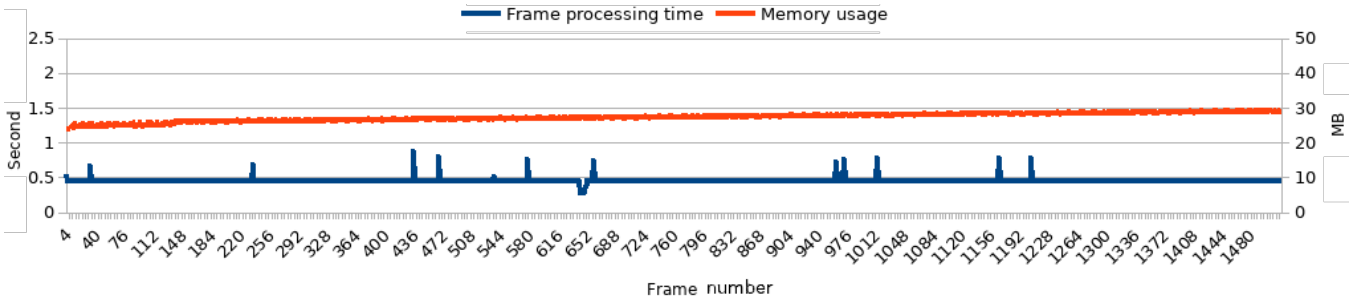


Figure 7. Original code base running on a tuned Ubuntu/Linux in a WCS. AVG energy consumption: 4.9J/frame.

is substantial variation in execution times and that some 59% of the frames miss their 500 ms deadline (blue line). The observed variation in execution time stems from the regular interference of the OS with various parts of the application software as well as hardware interrupts. We measure the energy of each frame by measuring the power consumption of the board using an external power supply (Kethley 2280S-32-6). This device allows us to measure at a frequency of 100 ms and link measurements with time stamps representing the beginning and end of the processing of each frame. We can then estimate the energy consumed while the frame was processed. During the frame processing time, other tasks of the system might be running, but since this is the case for all experiments, the average energy consumption per frame gives us a useful value for comparison.

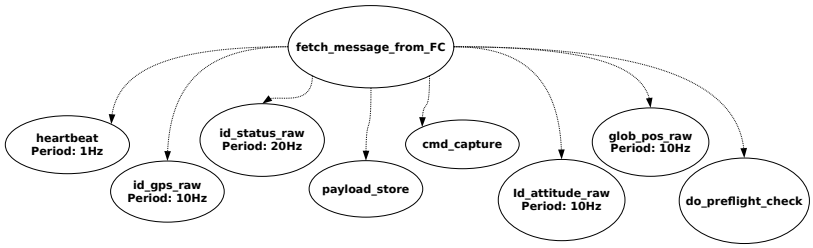
Since frames are queued before processing, missing the 500 ms deadline leads to increasing memory usage as images are added to the queue faster than they are processed (red line in Fig. 6). This results in increasingly delayed object detection reports until memory exhaustion eventually causes the whole application to crash. Since the real-time requirements of the application are not satisfied under worst-case conditions, we now apply the subsequent steps of the PReGO methodology.

4.2 Taming Stock Linux

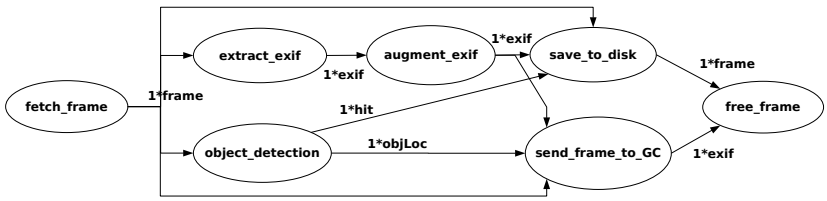
The second step of PReGO is taming Linux. As the presence of proprietary NVIDIA drivers prevents us from using the *PREEMPT-RT* patch set, we proceed with the five substeps introduced in Sec. 2.3. The first three taming steps are straightforward: we map all interrupt handlers to a specific CPU, disable real-time scheduler throttling, and disable all extraneous services. For step four, we store images and videos in separate storage devices. Originally, both images and videos are stored on one SD-card. However, the low writing speed of SD-cards causes interference when storing both video frames and images on the same device. To prevent resulting delays we attach an external USB drive to the board to save the video, thus removing the cause of the interference.

The last taming step is to disable the CPU governor, which is part of the Tegra driver installed in the OS. Migration points are beyond our control as the NVIDIA Tegra drivers are proprietary. We disable this governor by setting the value 0 in */sys/devices/system/cpu/cpuquiet/tegra_cpuquiet/enable* and enable all required cores for our application by setting the value 1 in all */sys/devices/system/cpu/*/online*.

Evaluation. Fig. 7 presents the result on the processing time and the memory usage after applying the five aforementioned customisation steps on our Ubuntu/Linux environment. We observe considerably less variation in frame processing times, but there are still 0.8% deadline misses. The energy consumption has increased from 3.8J to 4.9J, which we believe is primarily due to a combination of disabling the CPU governor and wasting significant CPU resources on thread management. We note that even though the CPU governor is disabled, the default NVIDIA dynamic frequency



(a) Task graph handling messages from the flight control module (Γ_{FC})



(b) Task graph processing a frame from the camera (Γ_{DLF})

(c) Task graph representing a GStreamer pipeline for capturing an image (Γ_{GI})

(d) Task graph representing a GStreamer pipeline for capturing a video (Γ_{GV})

Figure 8. Identified task graphs in the Search & Rescue application

scaling governor is still enabled. Therefore, high CPU load causes high energy consumption.

4.3 Task Model Identification

Applying existing schedulability analyses to our industrial use-case is not straightforward: The initial application provided by our partner was not designed to meet the underlying assumptions of such analyses. To enable existing analyses to be applied, we need to extract task graphs from the existing code and transform them to the task models supported by our methodology, as discussed in Sec. 2.4.

4.3.1 Task Identification. We extract task graphs from the initial system as described in Sec. 2.4. From the initial code we manually follow messages and frames variables, and we identify code sections that apply a specific feature on these variables. These portions of code are then identified as tasks while the exchanged messages/frames between these tasks create our task dependencies. From these tasks and dependencies we extract our task graphs and proceed with further PReGO steps.

The resulting task graphs are shown in Fig. 8, where each of the four sub-figures represents an independent Directed and Acyclic Graph (DAG). In the figure nodes represent tasks, and edges represent dependencies between tasks. Not all parameters are present for every node, as further discussed in Sec. 4.3.2. Each edge is labeled with the amount of data (a.k.a. number of *tokens*) that are transmitted along it and some type information about the data.

Fig. 8a shows the task graph that handles the reception of periodic messages sent by the Flight Control (later referred to as Γ_{FC}), previously discussed in Sec. 3.1. The source node/task of the graph first reads a message arriving on the serial port and then dispatches it to one of its successors for further processing. Hence, when a message is received, only one of the sinks is executed depending on the message content.

Fig. 8b represents the task graph that detects life boats at sea, later referred to as Γ_{DLF}. It first fetches a frame from a queue with atomic access. The frame is then sent to different tasks for further processing, such as Exchangeable image file format (Exif[12]) extraction and object detection. Upon successful detection, the number of detected objects is transmitted to one task (*save_to_disk*) while the object locations are sent to another task (*send_to_GC*). The sink task deallocates memory for the just processed frame and its corresponding *Exif* data.

Fig. 8c and 8d, later referred to as Γ_{GI} and Γ_{GV}, respectively, represent the two GStreamer pipelines used to capture frames from the HTTP server running on the Elphel board (see Sec. 3.2). The first one, Γ_{GI}, is in charge of fetching frames and storing them in a queue with atomic access, where they are later accessed by the task graph Γ_{DLF}. The second pipeline, Γ_{GV}, also fetches frames from the HTTP server, but to reconstruct the video of the mission that is stored on-board. Each pipeline is executed in a single thread, and all filters present in each pipeline run sequentially. With this in mind, and due to the lack of control we have over GStreamer library filters, we decided to represent each GStreamer pipeline as a task graph with a single task.

Having identified the task graphs in the system, we proceed by looking at their respective timing parameters. In the task graph Γ_{FC} some tasks have a period attached, whereas

[12]https://exifdata.com/

others do not. If a period is shown, we were able to extract the information from the flight control configuration (PX4 configuration provided by Sky-Watch[13], see Sec. 3.1). It corresponds to the receiving frequency of the respective message. For the sink nodes where no period is present the corresponding message is aperiodic. For example, the message triggering *payload_store* is sent only once to kill the payload application, whereas the message triggering *cmd_capture* is sent to activate/deactivate the image capturing and is therefore mission-dependent.

Due to the absence of timing constraints in the initial use-case, there are no explicit task-level deadlines. After further discussion and analysis, we decided to consider all task-level deadlines to be implicit. The consequence for the SAR application is that to process a new frame the previous one must be complete. The benefit of this decision is that it limits interference on shared resources, e.g. the GPU. However, it prevents us from exploiting frame-level parallelism.

Following the approach described in Sec. 2.4, we split the task graph Γ_{FC} into multiple periodic tasks. Regarding the two GStreamer tasks Γ_{GI} and Γ_{GV} we are not able to define any period, neither WCET nor WCRT. Therefore, we spatially isolate them (map on an unused core) and do not consider them in our analysis.

4.3.2 Inferring Periods.

In the task graph Γ_{FC} we have four tasks with a missing period. To enable the use of analyses for the periodic task model, we need to first infer this missing information, as described in Sec. 2.4. The tasks *payload_store* and *do_preflight_check* are aperiodic and will be grouped into a polling server meta-task. The task *cmd_capture* is also aperiodic as it responds to a message from flight control requesting to toggle the frame capturing. Similar to tasks *payload_store* and *do_preflight_check*, *cmd_capture* is placed in its own polling server as it requires a different period determined by the frame rate of the camera capture. As illustrated in Fig. 9, if the period for *cmd_capture* is too long then the task graph Γ_{DLF} (represented with its source node *fetch_frame* in the figure) will, in the worst case, miss the first frame, i.e. the life time ℓ of frame A is expired (ℓ depends on the frame rate, e.g. 2 fps implies $\ell = 500ms$).

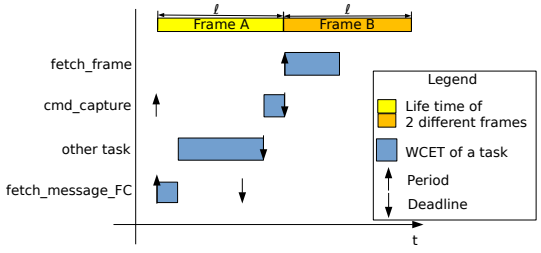


Figure 9. A *wrong* choice for task cmd_capture period leads to task fetch_frame missing the first frame

[13]www.sky-watch.com

To avoid the problem illustrated in the figure, the period of the server task must account for the reaction time upon message reception. Equation (1) limits the computed period to be the minimum period for a valid system. However, and as illustrated by the example of Fig. 9, in the unlikely event of other tasks with higher priority reaching their WCET, task *cmd_capture* can be delayed up to its WCRT. To overcome this situation, we make sure that the *cmd_capture* task gets a higher priority in our system. See Sec. 2.7.2 for details on priority assignment.

$$T_{cmd\_capture} \leq \ell - WCET_{fetch\_frame} - WCET_{cmd\_capture} - WCET_{fetch\_message\_FC} \quad (1)$$

$$T_{fetch\_msg\_FC} = \left\lfloor \frac{H}{\sum_{i \in successors(fetch\_msg\_FC)} \frac{H}{T_i}} \right\rfloor \quad (2)$$

The last missing period concerns the task *fetch_msg_FC*. This task periodically receives a message from the flight control and unicasts it to another task. To compute its period we compute the hyperperiod (H). From this, we derive the period as shown in Equation (2). Note that the derived period in Equation (2) is floored as time is in $\mathbb{N}^+$. This may result in one more activation of the task than strictly necessary. However, if there is no message to read from the flight control then the task simply completes earlier than its WCET without transmitting a message. Equation (2) holds because in this use-case we consider strictly periodic messages.

4.4 Task Re-engineering

Tasks and subtasks have now been identified and their timing properties extracted. We manually re-engineer the code of their implementations, as described in Sec. 2.5. We also describe the component coordination or task dependencies using the DSL from Fig. 2, but due to space limitations we cannot show the DSL code here, unfortunately.

4.5 Worst-case Execution Time

We estimate the WCET of each task using the previously described profiling-based technique. The sequential version of the application is generated and used to perform the measurements. The object detection algorithm includes both a CPU part and a GPU part, but, as described earlier, this is handled by stalling the CPU. We add a context switch overhead of $48ms$ to our measured WCET [14], which looks relevant to us as they cover ARM-based architectures running a Linux-based OS. The measured WCETs, including the 20% safety margin, are reported in Table 1.

Table 1. Mapping and priority table

Task name	WCET (ns)	Period (Hz)	WCRT (ns)	Priority	CPU ID
fetch_msg_FC	61333	*31.038*	*126916*	99	1
heartbeat	111165	1	*154831*	95	3
id_gps_raw	65583	10	*175165*	98	3
id_status_raw	21833	1	*21833*	93	3
cmd_capture	9358216	*11*	*9469381*	99	3
id_attitude_raw	22083	10	*39003917*	97	3
glob_pos_raw	16583	5	*113081*	96	3
payload_store do_preflight_...	48249 25583	*1*	*18938762*	94	3
fetch_frame	2682383			95	2
extract_exif	170414			96	1
object_detection	3374114			96	2
augment_exif	54333	2	*45283660*	97	1
save_to_disk	38981834			98	2
send_frame_...	150248			98	1
free_frame	23833			99	2

4.6 Scheduling

We show the resulting mapping and priority assignment in Table 1. We arbitrarily select *SCHED_FIFO*. Since one core is reserved for interrupts, the application tasks and the related schedulability analysis from Sec. 2.7 only make use of three of the four available cores (see Sec. 3.3).

4.7 Generating the Final Application

We use our toolchain to generate the final code, as described in Sec. 2.8. We generate one process per graph while each subtask becomes a thread of its corresponding process. Task management and communication follow the principle described in Fig. 3 with semaphores for releasing subtasks and shared-memory FIFO buffers for communication.

4.8 Evaluation & Validation

Fig. 10 presents the processing time and the memory usage after task identification using our toolchain and after enforcing the schedule on the application task graphs from Fig. 8. Task identification makes the parallelism more fine-grained, which improves performance, but, as shown in Fig. 10a, also decreases predictability without real-time scheduling.

Also, the per-frame energy consumption decreases significantly from the original code (4.9J) to our version with identified tasks (2.7J), but is again increased in the final version (3.1J). We believe the decrease to 2.7J is due to a better use of computing resources since the number of threads is increased and, thus, can better spread out the computation across the cores, letting the DVFS governor run some CPUs at lower speed. The increase to 3.1J is believed to be due to a different use of computing resources after reassigning threads to ensure real-time responsiveness of the system. In particular, threads were reassigned in a way that puts a significant computational load on each core of the system without fully loading any core of the system, which appears to increase the overall energy consumption of the system. Interestingly, this appears to indicate that dynamically changing the scheduling policy between real-time and non-real-time would provide a trade-off in energy consumption versus real-time behaviour. In the end, the per-frame energy consumption of 3.1J represents an 18% improvement over the initial 3.8J per frame (Fig. 6). For battery-powered devices like our drone this make a considerable difference.

The final version of the application from Fig. 10b was validated by Sky-Watch[14], deploying it to the actual drone platform and flying the default test mission, which concerns detecting a lifeboat placed in a uniform dry-land environment. The application was observed to perform identically to the original version from a functional point of view. Changes in the overall energy consumption of the drone platform (computation and actuation) were observed, but an analysis of the energy consumption of the system taking into account for external factors, such as changing wind conditions, is considered future work.

5 Related Work

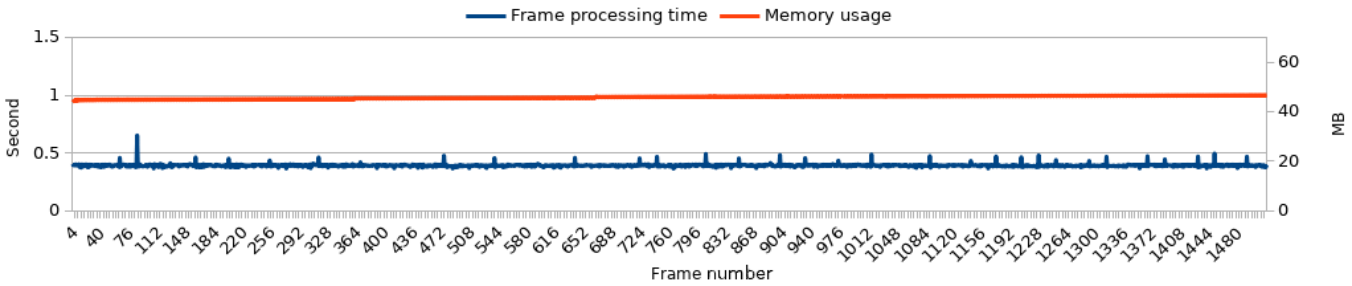
PReGO is based on the TeamPlay component-based architecture and declarative specification language, which were previously described in terms of general principles and the concept of energy-, time-, and security-aware scheduling [29]. This paper investigates the practice of applying this approach to a Linux-based system, presents the PReGO methodology for systematically applying TeamPlay to COTS-based systems, and is the first reported experimental evaluation of the TeamPlay component-based architecture and specification language.

The generative aspect of PReGO is based on model-driven software development (MDSD) [40], which is increasingly used in robotics [35] and drones [26, 31] to automatically generate parts of the implementation of a component-based system, such as ROS [28]. In the case of ROS, robotics toolchains like BRIDE [7] and SmartSoft [35] provide strong support for MDSD, and similarly rely on DSLs to specify the overall composition of the system. Unlike PReGO, these toolchains have, however, not explicitly been designed to tackle the challenge of integrating legacy code, making them better suited for developing new applications.

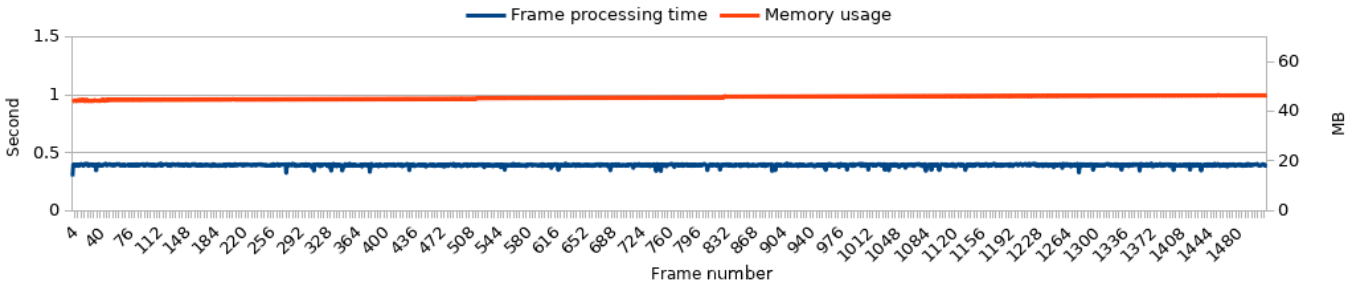
Drones have been the subject of research studies targeting many different use-cases. While most of them use COTS components, they mainly focus on the mission of the drone rather than extra-functional properties, such as guaranteeing the timing behaviour of the system. For example, [25, 39] concerns path planning, [17] focuses on communication with the ground station, [21] extracts environment features (e.g. water) from frames, and [11] focuses on object tracking.

In [12, 13] different applications are generated from a set of components. They use C++ and template programming capabilities to generate component code and glue code.

[14] www.sky-watch.com



(a) Version after task identification without real-time scheduling. Average energy consumption per frame: 2.7J



(b) Version after task identification with priority assignment and real-time scheduling. Average energy consumption per frame: 3.1J.

Figure 10. Final integration: componentisation and real-time scheduling

Similarly, [37] proposes a DSL to apply this generative metaprogramming paradigm to embedded systems. In [9], authors also propose a DSL and a generative framework along with a simulator, and targeting pervasive systems. Neither are these DSLs suitable for real-time systems, nor do they allow the timing analysis presented in this paper.

The language Lustre [4] and its compiler framework applies generative programming to embedded real-time systems. However, it is focused on the avionic domain, or more generally on control systems. In contrast, our methodology and framework covers a wider range of application domains, as the tool chain is independent of the inner body of tasks and only relies on the structure.

We acknowledge that using a proper RTOS like HIPPEROS [27] or an OS kernel with real-time capabilities like Linux with the PREEMPT_RT patch set or LitmusRT [10]) increases the control of the timing behaviour on the platform. However, we are constrained to use proprietary drivers and libraries not supported by such environments. Related work on taming Linux is mostly outdated, or the code is unavailable, e.g. implementation of EDF in the Linux kernel [16]. We found inspiration to tame our Linux kernel in [5, 23] for interrupt handling and scheduling enforcement and in [6] for CPU isolation.

In our work we deactivated the Dynamic Voltage and Frequency Scaling (DVFS) feature of the kernel to run cores at maximum frequency. Scordino et. al. on the contrary leave DVFS active in a Linux environment [32] and only deactivate it for real-time tasks to ensure timing constraints. This approach, however, relies on modifying the Linux kernel and its EDF scheduler (*SCHED_DEADLINE*), which was not an option in our setting.

6 Conclusion

This paper introduces a novel methodology called PReGO for satisfying real-time constraints on commercial-of-the-shelf (COTS) platforms with hardware and operating system software with generally unpredictable timing behaviour. PReGO includes both manual and generative steps. We illustrate our methodology on the industrial case study of a Search & Rescue application executing on a fixed-wing drone. Following PReGO, the use-case successfully satisfies its timing requirements in the worst-case scenario with no deadline misses. In addition, we managed to reduce the overall energy consumption from 3.8J to 3.1J per video frame processed by the application.

Our case study shows that PReGO successfully combines various engineering pieces into a systematic methodology that is well-suited to considerably increase trust in the real-time properties of applications executing on COTS platforms. This makes PReGO an attractive approach for (re-)engineering software in the large grey area of applications that need to meet real-time requirements, but are neither designed according to real-time principles nor are supposed to run on hardware and OS suitable for real-time guarantees.

To further improve the energy consumption and the predictability of the system, we plan to improve the task management and integrate a multi-mode scheduling policy. In addition, we plan to integrate our own energy-aware scheduler for the described task model.

References

[1] William B Ackerman. 1979. Data flow languages. In *International Workshop on Managing Requirements Knowledge (MARK)*. IEEE.

[2] Benny Akesson, Jozef Hooman, Jack Sleuters, and Adrian Yankov. 2020. Reducing design time and promoting evolvability using Domain-Specific Languages in an industrial context. In *Model Management and Analytics for Large Scale Systems*. Elsevier, 245–272.

[3] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. 2012. A generalized parallel task model for recurrent real-time processes. In *33rd Real-Time Systems Symposium (RTSS)*. IEEE.

[4] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* (2003).

[5] Björn B Brandenburg. 2011. *Scheduling and locking in multiprocessor real-time operating systems*. Ph.D. Dissertation. Univ. of Chapel Hill.

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

[7] Alexander Bubeck, Florian Weisshardt, and Alexander Verl. 2014. BRIDE-A toolchain for framework-independent development of industrial service robot applications. In *ISR/Robotik 2014; 41st International Symposium on Robotics*. VDE, 1–6.

[8] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. 2018. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *Real-Time Systems Symposium (RTSS)*. IEEE.

[9] Damien Cassou, Benjamin Bertran, Nicolas Loriant, and Charles Consel. 2009. A generative programming approach to developing pervasive computing systems. In *Proceedings of the eighth international conference on Generative programming and component engineering*. 137–146.

[10] Felipe Cerqueira and Björn Brandenburg. 2013. A comparison of scheduling latency in Linux, PREEMPT-RT, and LITMUS RT. In *9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. SYSGO AG.

[11] Peng Chen, Yuanjie Dang, Ronghua Liang, Wei Zhu, and Xiaofei He. 2017. Real-time object tracking on a drone with multi-inertial sensing data. *Transactions on Intelligent Transportation Systems* (2017).

[12] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. 2002. Generative programming for embedded software: An industrial experience report. In *International Conference on Generative Programming and Component Engineering*. Springer, 156–172.

[13] Krzysztof Czarnecki and Ulrich W Eisenecker. 1999. Components and generative programming. In *Software Engineering—ESEC/FSE'99*. Springer, 2–19.

[14] Francis M David, Jeffrey C Carlyle, and Roy H Campbell. 2007. Context switch overheads for Linux on ARM platforms. In *Proceedings of the 2007 workshop on Experimental computer science*.

[15] Robert I Davis, Liliana Cucu-Grosjean, Marko Bertogna, and Alan Burns. 2016. A review of priority assignment in real-time systems. *Journal of systems architecture* (2016).

[16] Dario Faggioli, Michael Trimarchi, Fabio Checconi, Marko Bertogna, and Antonio Mancina. 2009. An implementation of the earliest deadline first algorithm in linux. In *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM.

[17] Johannes Güldenring, Lucas Koring, Philipp Gorczak, and Christian Wietfeld. 2019. Heterogeneous Multilink Aggregation for Reliable UAV Communication in Maritime Search and Rescue Missions. In *International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE.

[18] Edward Ashford Lee and David G Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *Transactions on computers* (1987).

[19] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. 2018. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling* (2018).

[20] Chung Laung Liu and James W Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* (1973).

[21] Babak Majidi and Alireza Bab-Hadiashar. 2005. Real time aerial natural image interpretation for autonomous ranger drone navigation. In *Digital Image Computing: Techniques and Applications (DICTA)*. IEEE.

[22] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. 2015. WCET (m) estimation in multi-core systems using single core equivalence. In *27th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 3.

[23] Pedro Mejia-Alvarez, Luis Eduardo Leyva-del Foyo, and Arnaldo Diaz-Ramirez. 2018. *Interrupt Handling Schemes in Operating Systems*. Springer.

[24] Aloysius Ka-Lau Mok. 1983. *Fundamental design problems of distributed systems for the hard-real-time environment*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[25] Tobias Nägeli, Lukas Meier, Alexander Domahidi, Javier Alonso-Mora, and Otmar Hilliges. 2017. Real-time planning for automated multi-view drone cinematography. *ACM Transactions on Graphics (TOG)* (2017).

[26] Réda Nouacer, Mahmoud Hussein, Huascar Espinoza, Yassine Ouhammou, Matheus Ladeira, and Rodrigo Castiñeira. 2020. Towards a Framework of Key Technologies for Drones. *Microprocessors and Microsystems* (2020), 103142.

[27] Antonio Paolillo, Paul Rodriguez, Vladimir Svoboda, Olivier Desenfans, Joël Goossens, Ben Rodriguez, Sylvain Girbal, Madeleine Faugere, and Philippe Bonnot. 2017. Porting a safety-critical industrial application on a mixed-criticality enabled real-time operating system. In *5th Workshop on Mixed Criticality Systems (WMC), RTSS*.

[28] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[29] Julius Roeder, Benjamin Rouxel, Sebastian Altmeyer, and Clemens Grelck. 2020. Towards Energy-, Time- and Security-Aware Multi-core Coordination. In *Coordination Models and Languages, 22nd International Conference*. Springer, LNCS 12134.

[30] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. 2019. Hiding communication delays in contention-free execution for SPM-based multi-core architectures. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[31] Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Massimo Tivoli. 2016. Automatic generation of detailed flight plans from high-level mission descriptions. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 45–55.

[32] Claudio Scordino, Luca Abeni, and Juri Lelli. 2019. Real-time and energy efficiency in Linux: theory and practice. *Applied Computing Review* (2019).

[33] Paul F Smith, Sameer M Prabhu, and Jonathon Friedman. 2007. *Best practices for establishing a model-based design culture*. Technical Report. SAE Technical Paper.

[34] Sundararajan Sriram and Shuvra S Bhattacharyya. 2018. *Embedded multiprocessors: Scheduling and synchronization*. CRC press.

[35] Andreas Steck, Alex Lotz, and Christian Schlegel. 2011. Model-Driven Engineering and Run-Time Model-Usage in Service Robotics. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering* (Portland, Oregon, USA) *(GPCE '11)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2047862.2047875

[36] Martin Stigge and Wang Yi. 2015. Graph-based models for real-time workload: a survey. *Real-Time Systems* (2015).

[37] Janos Sztipanovits and Gabor Karsai. 2002. Generative programming for embedded systems. In *International Conference on Generative Programming and Component Engineering*. Springer, 32–49.

[38] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Compiler Construction*. Springer.

[39] Janis Tiemann, Ole Feldmeier, and Christian Wietfeld. 2018. Supporting maritime search and rescue missions through UAS-based wireless localization. In *Globecom Workshops (GC Wkshps)*. IEEE.

[40] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. 2013. *Model-driven software development: technology, engineering, management*. John Wiley & Sons.

[41] Zheng Wang and Michael FP O'Boyle. 2010. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *19th International Conference on Parallel Architectures and Compilation Techniques*. ACM.

[42] Jon Whittle, John Hutchinson, and Mark Rouncefield. 2013. The state of practice in model-driven engineering. *Software* (2013).

[43] 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. *Transactions on Embedded Computing Systems (TECS)* (2008).