Bachelor's Thesis
May 2021

Thermal and Power Management in Avionics OS Scheduler

Matěj Kafka
dev@matejkafka.com

Supervisor: Ing. Michal Sojka, Ph.D.

Department of Cybernetics
Faculty of Electrical Engineering

Abstract

High-performance MPSoCs are becoming more common in the avionics and automotive industries for safety-critical, real-time applications, bringing new challenges in scheduling and thermal management. However, some cutting-edge computing platforms like the i.MX8QuadMax by NPX used for my research are not yet supported by the proprietary real-time operating systems (RTOS) used in the industry.

To allow for prototyping and benchmarking of avionic workloads on such platforms and development of new thermal-aware scheduling algorithms, we developed DEmOS, an open-source static scheduler running in Linux user-space, mimicking a RTOS scheduler using existing Linux user-space interfaces. DEmOS was already successfully used to evaluate a new thermal-aware scheduling algoritm.

https://github.com/CTU-IIG/demos-sched/

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Kafka Matěj** Personal ID number: **483777**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Thermal and Power Management in Avionics OS Scheduler

Bachelor's thesis title in Czech:

Správa teplot a napájení v rozvrhovači avionického operačního systému

Guidelines:

1. Make yourself familiar with DEmOS – a Linux-based simulator of avionics OS scheduler for safety-critical applications. Also get familiar with ARINC 653 specification of avionics RTOSes.
2. Prepare a Linux kernel for a board with i.MX8 QuadMax CPU. Take the available Yocto distribution as a basis, but allow independent compilation. Try to resolve the kernel problems that appear when used with DEmOS – either by upgrading the kernel or patching/modifying the relevant parts of it. Ensure that the GPU is supported by the new kernel.
3. Extend DEmOS to support application initialization and to incorporate mechanisms for thermal and power management (DVFS, sleep states etc.) for i.MX8 platform. Implement several thermal/power management policies on top of the developed mechanisms.
4. Evaluate the mechanisms and policies on both synthetic and real-world benchmarks. Besides CPU applications, consider also GPU applications (OpenCL). If possible, evaluate thermal and power properties of OpenCL applications when running either on the GPU or on the CPU. Also try to evaluate the overhead (latency) of CPU power state transitions.
5. Document the results thoroughly.

Bibliography / sources:

[1] DDC-I, Inc., "Deos, a Time & Space Partitioned, Multi-core Enabled, DO-178C DAL A Certifiable RTOS – DDC-I." https://www.ddci.com/products_deos_do_178c_arinc_653/
[2] THERMAC Project, „Preliminary implementation of a thermal□aware resource management policy", Deliverable D4.1, 2020
[3] ARINC specification 653P1-2, Avionics Application Software Standard Interface, Aeronatical Radio, Int., 2006
[4] https://github.com/adubey14/arinc653emulator

Name and workplace of bachelor's thesis supervisor:

Ing. Michal Sojka, Ph.D., Embedded Systems, CIIRC

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **25.01.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

Ing. Michal Sojka, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

Abstrakt (CZ)

V leteckém a automobilovém průmyslu je stále obvyklejší používání výkonných vícejádrových čipů pro aplikace realného času s vysokými bezpečnostními požadavky, přinášející nové výzvy v oblasti rozvrhování a řízení teplot. Nové výkonné platformy, např. NXP i.MX8QuadMax, kterou používáme pro náš výzkum, však zatím nejsou podporovány typicky používanými proprietárními operačními systémy reálného času (RTOS).
Pro prototypování a testování avionických systémů na těchto platformách a vývoj nových rozvrhovacích algoritmů jsme vyvinuli open-source nástroj DEmOS, statický rozvrhovač bežící v user-space na Linuxu, který napodobuje RTOS rozvrhovače s využitím již existujících systémových rozhraní. DEmOS byl již úspěšně využit pro testování nového rozvrhovacího algoritmu zohledňující výsledné teploty.

Acknowledgement

I would like to thank my supervisor for always being fair, upfront and helpful, and my colleagues Jakub Dupák, Max Hollmann and Vojtěch Štěpančík for many discussions on code architecture, mental support and our shared drive to always do better.

I would also like to acknowledge the initial work on DEmOS done by Ing. Jiří Záhora and my supervisor, Ing. Michal Sojka, Ph.D., which I expanded upon over the past year.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 21. 5. 2021

.....

Table of contents

1 Introduction

In avionics and automotive domains, there is now a strong demand for high-performance Multi-Processor Systems-on-Chip (MPSoC), due to a rising use of digital image processing and machine learning applications [2], and consolidation of system components to a smaller number of hardware modules (Integrated modular avionics [1]). For some of these uses, GPUs are also vital.

These systems operate under harsh environmental conditions such as dust, vibration and extended thermal ranges. To ensure safety and reliability of the system, it is vital to operate within a thermal envelope. However, in avionics, thermal management has been a long-term problem even for lower-performance platforms [3], and it is further exemplified by the higher thermal emission of more powerful MPSoCs, and especially GPUs.

One possible solution is the use of active cooling, which brings issues with mechanical design and increases weight and costs. An alternative way is to employ passive cooling techniques directly on the platform, such as dynamic voltage and frequency scaling and thermal-aware scheduling, which is the approach our research group focuses on.

To evaluate the developed models and scheduling algorithms on real hardware, we need an execution environment similar to the one provided by a safety-critical, real-time operating system (RTOS). Many cutting-edge high-performance platforms are not yet supported by the RTOSs commonly used and required in both industries, such as the i.MX8QuadMax by NXP, which is targeted by our research. Usually, Linux is the first operating system supported on these platforms; however, its runtime environment differs significantly from the one provided by an RTOS, primarily in the scheduling model and provided temporal guarantees.

To allow for prototyping and benchmarking of avionic workloads on such platforms and development of new thermal-aware scheduling algorithms, we **developed an open-source tool called DEmOS** [5], a static scheduler running in Linux user-space, mimicking a RTOS scheduler using existing Linux user-space interfaces.

The rest of this thesis presents the design, implementation and performance evaluation of DEmOS. The project was originally started by Ing. Jiří Záhora and my supervisor, Ing. Michal Sojka, Ph.D., who developed a working prototype. I took over the project, extended it with new functionality, primarily the power management system, and updated many of the existing components to be more robust and extensible. The goal of my work was to further develop DEmOS as a tool for other researchers, and further research is not the subject of this thesis.

2 Background

This section provides an overview of the topics required in later sections. First, real-time and safety-critical systems are described, including relevant standards. Then, the Linux kernel features used by DEmOS are briefly introduced.

2.1 Real-time systems

In most consumer-facing software systems, emphasis during design and development is placed on functionality and its correctness; in cases where "performance" is considered, hardware resource usage and total throughput of the system is typically prioritized over other parameters like latencies, and no hard timing constraints are present.

In contrast, for many domains of embedded development, throughput is generally not a prioritized metric, with reliability, deterministic timing and bounded latencies being the more important design goals. [6] One example of such a system is the flight control module of a helicopter — to keep the aircraft stable, a set of tasks must be ran periodically, and missing any of them due to a temporary lockup may result in instability or even a crash. Specification of such a system includes both logical (system gives correct results) and temporal (the results are delivered at the correct time) requirements, and both must be met for the system to be usable. We call these **real-time systems**.

Perhaps somewhat counterintuitively, low latencies are not necessarily correlated with high throughput. A common example in software engineering is the use of buffers and caches — if used well, both improve throughput and even average latency, but the worst-case end-to-end latency, an important metric for real-time systems, is typically negatively impacted. In general, real-time systems tend to rely on simple, deterministic components, as they are easier to reason about and verify temporal guarantees for. [7]

We distinguish two types of real-time systems: hard real-time and soft real-time. If a deadline has to be met under all circumstances, it is called a **hard deadline**; a **hard real-time system** is a system where all deadlines are hard. Here, a failure to meet a deadline is often catastrophic. Analysis of such systems is easier, as the system can be considered fully deterministic. The aforementioned example of an aircraft flight control is a hard real-time system; hardware control loops in general are typically hard real-time systems — an autonomous driving system, a print head controller, a pacemaker, etc. These systems are implemented on top of specialized real-time operating systems

(RTOS), as general-purpose operating systems (GPOS) are too complex and their timing characteristics often too unpredictable to reason about.

Systems where some deadlines may occasionally be missed are called **soft real-time systems**. Here, the full latency distribution is considered, not only the worst-case behavior—a deadline should be met "most of the time", otherwise the performance deteriorates. This makes the system analysis more complicated, due to its probabilistic nature. A common example is audio processing and playback — a missed deadline results in perceptible stuttering, but as long as it occurs infrequently, it is acceptable (or rather, designing the system to be hard real-time would be too expensive given the complex algorithms commonly used). Deciding whether a system should be hard or soft real-time is for many domains up to the designer, and typically involves a trade-off between reliability, throughput, development time and costs.

2.1.1 CPU scheduler

A CPU scheduler is part of an operating system that switches between ready tasks so that they can all run concurrently. Schedulers used in GPOSs are typically geared towards throughput, fairness (avoiding CPU starvation of any task) and, for desktop, interactive use (prioritizing a few tasks the user is actively interacting with). [8] In order to efficiently distribute computing time in a highly dynamic environment of a typical server or desktop system, the scheduling algorithms are quite complex, making it hard to reason about and make any temporal guarantees.

At the same time, the kernel and drivers in GPOS are typically not designed with hard real-time applications in mind, and fundamental changes to the system would be required to support these, usually sacrificing some throughput and interactivity. Kernel and driver preemptibility is one such issue, with some portions of the mainline Linux kernel still not preemptible (although the `CONFIG_PREEMPT_RT` option improves the situation) [9], and Windows NT kernel fully preemptible, but with some of the proprietary drivers blocking DPC queues for longer periods of time, also making it unsuitable for real-time applications. One solution that avoids these issues is using a hard real-time hypervisor that runs the GPOS as a fully preemptible guest OS.

For an RTOS scheduler, the main requirement is predictability — the scheduler behavior must be deterministic, allowing the system designer to reason about the worst-case temporal behavior of each task. For an RTOS, the environment is typically mostly static and the system designer has control over all tasks — the complexity and the resulting unpredictability of a GPOS scheduler is therefore not desirable.

2.1.2 Real-time CPU scheduling algorithms

The goal of a real-time CPU scheduling algorithm is to distribute CPU time (and other resources) to tasks in a way so that all deadlines are met. Three types of tasks are scheduled: periodic tasks, which must meet a periodic deadline (an instance of the task must finish in each period); sporadic tasks, which arrive at arbitrary times and have a

hard deadline; and aperiodic tasks, which also arrive at arbitrary times, but without any hard deadline.

Two classes of scheduling algorithms are typically used: static (offline, clock-driven) scheduling and dynamic (online) scheduling.

Static scheduling

With static scheduling, the full schedule for periodic tasks is computed offline (at design time). At runtime, the scheduling decisions are only done at predefined time instants by switching tasks according to the schedule. This way, runtime overhead is minimal, and more complex algorithms may be used to find an optimal schedule ahead-of-time. As the full schedule is known at design time, it is easier to verify.

On arrival of a sporadic task, an admission test may be performed, and if the pending task cannot be feasibly scheduled, it is rejected; otherwise, it is executed during free intervals in the schedule. For an aperiodic task, it is queued and ran when the system would otherwise be idle (with possible optimizations like slack stealing).

The disadvantage of static scheduling is its inflexibility, as the schedule is fixed and all tasks and their parameters must be known at design time.

Dynamic scheduling

With dynamic scheduling, scheduling decisions are done at runtime, depending on which tasks are currently ready to be executed. This lets the system react better to sporadic and aperiodic tasks, at the cost of higher overhead and more complex verification at design time.

A simple example is the FCFS ("first come, first serve") scheduling algorithm, which keeps a queue of all ready tasks and schedules them in the order of arrival.

More complex dynamic schedulers typically assign a priority to each task, and give CPU time to the highest-priority task out of all ready tasks. Task priorities are either **fixed** (set at design time) or **dynamic** (set at runtime, often based on the task deadline). As DEmOS only uses a very basic form of FCFS scheduling, these will not be explored further — for more details, refer to [6].

2.2 Safety-critical systems

Safety-critical systems are systems which have defined safety requirements, which must be fulfilled for the system to perform acceptably.

For many non-mechanical (e.g. software) systems used by consumers, an occasional failure or malfunction is acceptable, although not desirable, as ensuring correct functionality under all circumstances would be too complex and/or expensive, the impact of a failure is minor, and it is easy enough for the operator to manually recover the system back to a valid state. We will call these **best-effort systems** (also called "non-critical"). One example is the audio processing and playback system already mentioned above, which is also a soft real-time system.

In comparison, a failure or malfunction in a **safety-critical** system may result in injury, death, severe material or financial loss or other outcomes deemed unacceptable by the system designer. This is common in cases where physical machinery is part of the system, e.g. flight control in an airplane, or a medical ventilation system. Safety-critical systems also typically have real-time requirements, but this is not always true — for example, the correctness of a compiler output is safety-critical, but no real-time requirements are present.

The safety requirements for such software systems are significantly stronger than in typical software development — the system developers must be able to assert safety guarantees within a rigorous theoretical framework; this is the subject of system safety engineering. During development, possible hazards should be documented, analyzed and the system designed with mitigation measures to minimize the probabilistic risk of failure — all hazards should either be eliminated, or probabilistic in nature, with the associated risk reduced to an acceptable level.

There are multiple commonly used standards, sets of best practices and safety analysis procedures, some of them specific to the avionics domain. Unlike with many other software development domains, standard compliance is required for most commercial usage and ensured using a stringent and typically expensive certification process.

Some of the better known system safety standards are:

- DOD MIL-STD 882E [10], a top-level system safety standard typically required in contracts for the USA Department of Defense, defining common terminology and safety requirements throughout the full life cycle of system development and maintenance
- IEC 61508, a generic international safety standard, applicable to all industries
- MISRA C, a set of software development guidelines for the C language, aiming to make the language safer for use in automotive industry

- DO-178C / ED-12C, a primary guideline by which certification of all commercial software-based avionic systems in the USA and Europe for civil (non-military) use is done
- ISO 26262, an international standard for safety-critical systems installed in road vehicles

As it is not feasible to certify a large monolithic operating system due to the code size and complexity, specialized microkernels are used for safety-critical systems. Some commonly used certified safety-critical operating systems are VxWorks by Wind River, Deos by DDC-I and QNX by Blackberry. All the mentioned operating systems also fall into RTOS category.

Commonly, an integrated system has both safety-critical and non-critical components. As the level of scrutiny and correctness guarantees is typically lower for the non-critical components, it is important to isolate parts of the system with differing safety levels to constrain the scope of potential failure. At software level, the ARINC 653 standard specifies means to achieve such isolation.

2.3 ARINC 653 standard

ARINC 653 [11] is a software specification, prescribing the baseline operating environment in the context of safety-critical RTOSs. It specifies an interface between the operating system and applications called Application Executive (APEX), together with the communication and scheduling model and isolation requirements on memory, CPU and I/O. It allows for safe consolidation of multiple independent modules of different safety levels on the same hardware — an important part of the IMA architecture, allowing for more efficient hardware resource use, weight savings, design simplification and easier maintenance. For our purposes, we will focus on the isolation and scheduling requirements.

A failure in one module must not impact other modules, equivalently to running the modules without any shared resources. Real-time properties of each module should also be preserved, independently of the rest of the system, which may change during normal operation (e.g. in avionic context, a module may only be active during a take-off and landing).

To minimize interference, each software module runs in an isolated partition. This is achieved by providing a separate virtual memory space for each partition (spatial isolation) and assigning other resources (CPU, GPU, I/O,...) according to a fixed schedule (temporal isolation), where only a single partition runs at a time. The original ARINC 653 standard published in 2010 did not address its use in multi-core processor systems; due to a growing market demand, however, the 2015 update to the standard now supports parallel execution of partitions on multiple cores.

Each partition may contain multiple processes, which may optionally share the same memory space, and which are typically scheduled using a fixed-priority dynamic scheduler, local to each partition, as opposed to the global static scheduler. This allows for more efficient resource sharing inside the partition, while still supporting hard real-time applications. Each partition contains an initialization process, which is responsible for setting up the partition, including other processes. Dynamic memory allocation is allowed, but only from a statically defined pool, so that the upper bound on used memory is known at compile time.

A multi-level health monitoring system is also specified, with high-priority error handling processes that allow the system to recover from error states (memory access violation, deadline miss,...), both on partition and module level.

The system is configured using an XML configuration file with a specified schema, which should be portable between all compliant systems, so that the same applications can be used on different platforms without code changes, and, therefore, without the need for recertification.

There are 2 common approaches to implementing an ARINC 653 compatible system — a hypervisor-based solution, and a kernel-based one.

2.3.1 Hypervisor-based implementation

A hypervisor is used to implement scheduling, resource access control and APEX. [12] This way, it is possible to run a full operating system inside a partition. System designers may use a certified RTOS for safety-critical modules, and a better supported OS like Linux for best-effort modules, e.g. a car infotainment system — this allows using common, more full-featured libraries. At the same time, the hypervisor only implements core functionalities, which results in a smaller codebase, where safety and security are easier to verify, resulting in a more reliable system with lower certification costs.

2.3.2 Kernel-based implementation

Typically, an existing safety-critical RTOS kernel is modified to support ARINC-653 compatible scheduling, and provide the APEX interface. This approach may typically offer better performance, but it is less flexible and harder to verify, resulting in higher certification costs and lower reliability.

2.4 Deos™ RTOS

Deos™ [13] by DDC-I is one of the popular RTOSes in the avionics domain. Deos™ supports ARINC 653 time and space partitioning and has been certified in numerous safety-critical products to DO-178 DAL-A.

In each partition, Deos™ RTOS supports one of the following three schedulers: harmonic Rate Monotonic, ARINC 653, and POSIX (leveraging a para-virtualized RTEMS instance). In a multi-core processor, a single Deos™ RTOS instance handles all processor cores. A window has an ID, a fixed length, and spawns across all cores. Each core within the window has its own scheduler (e.g. ARINC 653, POSIX). User applications (processes with threads) are mapped to the windows, cores, and schedulers during the system configuration.

In the context of an IMA architecture, Deos™ provides a set of mechanisms that allow a single multi-core platform to host multiple applications of different criticality levels. In many cases, just two levels are used: Safety-Critical (SC) and Best-Effort (BE). Deos™ extends the ARINC 653 scheduling scheme by allowing SC and BE partitions to share a core in a window. Specifically, SC partitions are always granted for execution within a given window and BE partitions can be optionally scheduled once all SC partitions in the window complete.

2.5 Linux *cgroup*

cgroup [14] (abbreviated from "control groups") is a Linux kernel mechanism to organize processes hierarchically and distribute system resources along the hierarchy in a controlled and configurable manner. For example, it may be used to suspend a group of processes, restrict allowed CPU cores and used memory or monitor spawned child processes.

cgroup is largely composed of two parts - the core and controllers. *cgroup* core is primarily responsible for hierarchically organizing processes. A *cgroup* controller is usually responsible for distributing a specific type of system resource along the hierarchy although there are utility controllers which serve purposes other than resource distribution.

cgroups form a tree structure and every process in the system belongs to one and only one cgroup. All threads of a process belong to the same cgroup. On creation, all processes are put in the cgroup that the parent process belongs to at the time. A process can be migrated to another cgroup. Migration of a process does not affect already existing descendant processes.

The original version of *cgroup* was included into mainline Linux kernel in 2007; this version is now called *cgroup v1*. Between 2013–2016, *cgroup* was redesigned and now only uses a single, unified hierarchy, with updated controller interfaces; this version is called *cgroup v2*. This version significantly simplifies cgroup management for both kernel and user-space.

2.5.1 *cgroup* modes (*systemd*)

Before using, *cgroup* virtual filesystem hierarchy must be mounted. This is almost always done automatically by the *init* system during the startup, which is, for mainstream distributions, typically *systemd*, which mounts the hierarchy in one of three possible modes: [15]

- Unified — a modern *cgroup v2* mode, where a single unified hierarchy is mounted as `/sys/fs/cgroup`; this is the only actively developed mode, supporting all *cgroup v2* features.
- Legacy — a legacy *cgroup v1* hierarchy, where each controller is mounted separately as `/sys/fs/cgroup/<controller>`. This mode is not actively supported, only exists for backwards compatibility and no major Linux distributions use it by default.
- Hybrid — this mode is similar to legacy mode, but an extra `/sys/fs/cgroup/unified` hierarchy is mounted, which exposes the core *cgroup v2* functionality. This mode is also not actively supported, but up until recently, most Linux distributions defaulted to it, as Docker and other container implementations, which use *cgroup* for core functionality, did not support the unified *cgroup v2* hierarchy. However, Docker added support in late 2020 and Fedora 31, released in 2019 already defaults to unified mode, with rumors of other distributions possibly following suit in the near future.

2.6 CPU performance scaling

The majority of modern processors are capable of operating in a number of different clock frequency and voltage configurations, often referred to as P-states. [16] As a rule, the higher the clock frequency and voltage, the more instructions can be retired by the CPU over a unit of time, but also more power is drawn over a unit of time by the CPU in the given P-state. Therefore, there is a natural tradeoff between the CPU capacity (the number of instructions that can be executed over a unit of time), and the power drawn by the CPU.

In some situations it is desirable or even necessary to run a program as fast as possible. In that case, there is no reason to use any P-states different from the highest one (i.e. the highest-performance frequency/voltage configuration available). In some other cases, however, it may not be necessary to execute instructions so quickly and maintaining the highest available CPU capacity for a relatively long time without utilizing it entirely may be regarded as wasteful. It also may not be physically possible to maintain the maximum CPU capacity due to thermal constraints, power supply capacity or similar. To cover those cases, there are hardware interfaces allowing CPUs to be put into different P-states, i.e. switched between different frequency/voltage configurations.

Typically, these interfaces are used along with algorithms to estimate the required CPU capacity, so as to decide which P-states to put the CPUs into. Since the utilization of the system generally changes over time, that has to be done repeatedly on a regular basis. The activity by which this happens is referred to as CPU performance scaling or CPU frequency scaling (as it involves adjusting the CPU clock frequency).

For commonly used CPU architectures, the frequency cannot be varied continually, but only at fixed steps. For desktop CPUs, the frequency selection is usually quite granular, with steps in the order of 100 MHz. In smaller CPUs used for embedded and mobile development, commonly only a small set of possible frequencies is available, with multiple cores in a cluster often sharing the same P-state.

2.7 Linux *CPUFreq*

CPUFreq [16] is a Linux kernel subsystem that controls CPU frequency scaling, allowing the system to balance performance with power consumption. Note that the term "CPU" is used for logical CPUs here — a single physical processor may contain multiple logical CPUs, commonly referred to as "CPU cores".

The *CPUFreq* subsystem consists of three layers: the core, scaling governors and scaling drivers. The core provides the basic framework and user-space interfaces. Scaling governors each implement a scaling algorithm, varying the active P-state as required depending on system load. Scaling drivers communicate with the hardware, provide information on the available P-states and physically change the CPU P-states according to the active scaling governor.

Depending on the processor model and kernel configuration, one of multiple available *CPUFreq* drivers is loaded during boot. Most drivers have a similar interface, except the *intel_pstate* driver [17], which is used on Intel CPUs — here, frequency scaling is done internally by the processor and the driver. To use generic scaling governors, the driver must be switched to the so-called passive mode, where automatic hardware scaling is disabled.

On some processors, multiple CPUs share the same selected P-state. To represent this, *CPUFreq* has the so-called policies, which represent a set of CPUs with shared parameters. To prevent confusion with the concept of power policy used in DEmOS, these will be referred to as "*CPUFreq policy*" in further sections.

2.7.1 *CPUFreq* user-space interface

CPUFreq is, when supported, available to user-space as part of the `sysfs` virtual filesystem, typically accessible under the `/sys/devices/system/cpu/cpufreq` directory. For each existing *CPUFreq policy*, a subdirectory `policy<n>` exists, where `<n>` is an integer index. CPUs affected by this policy can be listed by reading the `affected_cpus` virtual file inside the subdirectory. The active governor for a policy is

configured using the `scaling_governor` file, with available values listed in `scaling_available_governors` file.

Most governors only expose general parameters like min/max frequency, except for the `userspace` governor, which allows a user-space client to select a precise P-state for each *CPUFreq policy*, and keeps the selected P-state until requested otherwise, unless the thermal envelope is exceeded, which may occur with, for example, the Turbo mode of Intel processors.

When `userspace` governor is selected, current P-state may be selected by writing the desired frequency in the `scaling_setspeed` file. *CPUFreq* uses frequencies to represent the available P-states, but voltage and other parameters are varied accordingly.

To set P-states manually on a system where *intel_pstate* driver is used, the driver must be first switched to passive mode by writing to the `/sys/devices/system/cpu/intel_pstate/status` virtual file. Then, it behaves as the generic driver and allows clients to use the `userspace` governor.

3 Design

This section describes the external behavior of DEmOS and also provides a high-level overview of internal design. It contains useful information both for the end user and a contributor.

3.1 Overview

DEmOS is a user-space Linux program emulating an avionics real-time scheduler using existing Linux kernel features. In addition to scheduling, DEmOS also provides a power management subsystem, allowing users to define custom power policies.

Design and terminology used is inspired by the Deos™ operating system, which can be configured to run in ARINC 653 compatible mode. DEmOS itself is not ARINC 653 compatible, neither it attempts to be, although some components are implemented as specified. It does not implement the APEX interface, opting to let processes use the native Linux API, and only focuses on scheduling. What it strives to preserve is the scheduling model, so that the thermal and power characteristics of avionic workloads may be analyzed and improved upon using DEmOS.

An `epoll`-based event loop is used to receive and process events on a single thread. Linux *cgroup v2* is used to control the scheduled processes. CPU frequency scaling is adjusted using the `sysfs`-based *CPUFreq* interface.

3.2 Scheduling model

DEmOS manages **partitions**, which are groups of **processes**. Partitions are scheduled according to a static schedule defined by the user. Similarly to ARINC 653, time is divided into **time windows** of fixed length, providing temporal isolation. Each window contains a set of **slices**, which bind a partition to a subset of the available CPUs for the duration of the time window. Slices provide space partitioning inside a time window, allowing multiple partitions to execute in parallel on different CPUs. DEmOS schedules all defined windows in an interval called a **major frame**, which repeats periodically. A window execution examples are shown on Fig. 1 and Fig. 2.

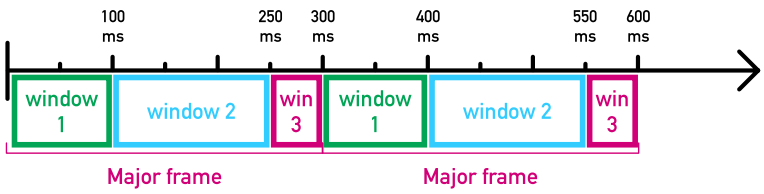

Fig. 1. Periodic execution of the defined time window sequence.

Processes inside a partition are scheduled in a fixed predefined sequence, without a static schedule. Next process is started either when the previous process exhausts its time budget for the current window, or when it voluntarily yields after completing all work required in the current window. DEmOS exits when either all processes end or the scheduler receives a stop signal. See Fig 3 for an example execution diagram.

Two types of partitions are supported: safety-critical and best-effort. **Safety-critical partitions** are started at the beginning of a window, and all processes must complete before the end (otherwise, it is a configuration error). **Best-effort partitions** are optionally executed in the remaining free part of a window after the safety-critical partition finishes, and automatically preempted at the end of the window. DEmOS may skip execution of the best-effort partition if required by the power policy [section 3.6] due to thermal constraints.

Each slice contains at most one safety-critical partition, followed by an optional best-effort partition. DEmOS supports 2 modes of scheduling — either the best-effort partition is launched immediately after the corresponding safety-critical partition finishes, or all best-effort partitions wait until the last safety-critical partition in the whole window finishes, which minimizes interference between safety-critical and best-effort partitions.

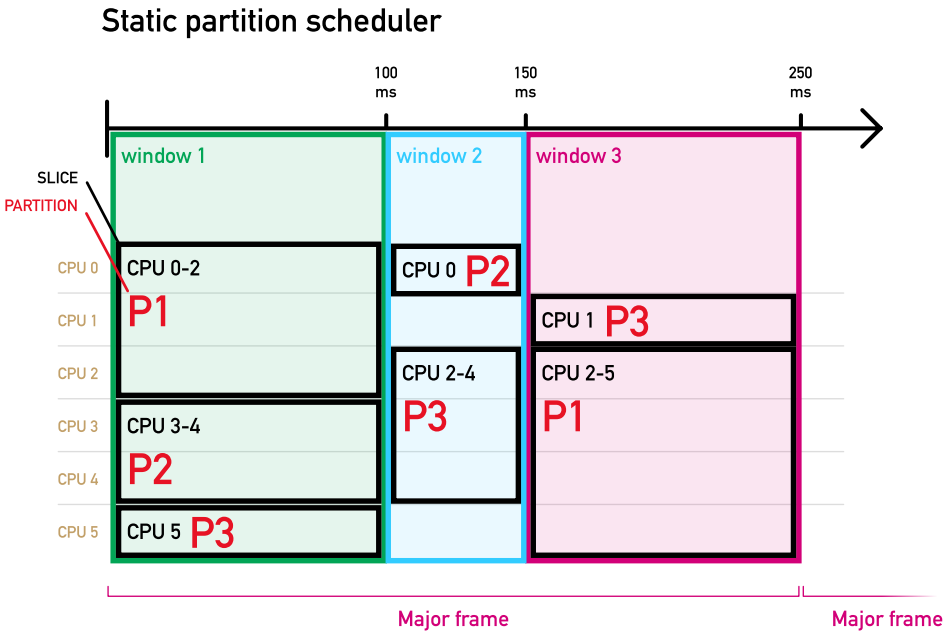

Fig. 2. A time diagram illustrating the behavior of partition scheduler.

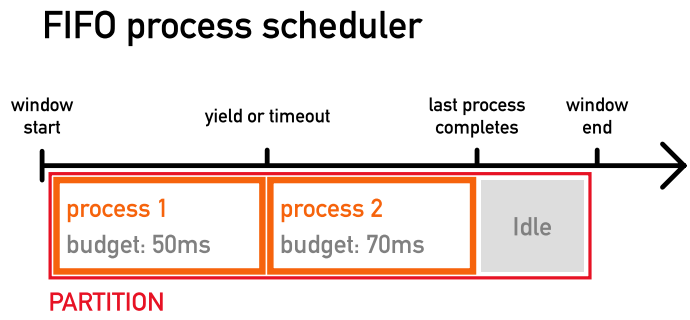

Fig. 3. A time diagram illustrating the behavior of process scheduler.

3.3 Configuration

DEmOS is configured using two different interfaces: a YAML-based **configuration file** and a **command-line interface (CLI)**. The configuration file defines the partitions and the static schedule. The CLI then supplements the configuration file via various additional runtime options such as power policy, frame synchronization messages and logging.

Most CLI options are described in the following sections, with full specification available in Appendix 1. The full format of the configuration file is specified in Appendix 2.

```
1 set_cwd
2 partitions: [name, processes: [{cmd, budget, jitter, init}]]
3 windows: [length, slices: [{cpu, sc_partition, be_partition}]]
```

Fig. 4. A simplified schema of the YAML-based configuration format.

3.4 Example run

To illustrate the introduced concepts, see the following example configuration file, together with a trace of the execution, recorded using `trace-cmd` and visualized with `kernelshark`.

```yaml
windows:
  - length: 200
    slices:
      - {cpu: 0, sc_partition: SC1, be_partition: BE1}
      - {cpu: 1, sc_partition: SC2}

partitions:
  - name: SC1
    processes:
      - {budget: 100, cmd: proc_SC1-1}
      - {budget: 50, cmd: proc_SC1-2}
  - name: BE1
    processes:
      - {budget: 25, cmd: proc_BE1-1}
  - name: SC2
    processes:
      - {budget: 175, cmd: proc_SC2-1}

```

Fig. 5. An example DEmOS configuration file.

The configuration defines 3 partitions containing 4 processes, which are executed in a single window. Processes in partitions `SC1` and `BE1` execute on CPU 0, and the single process inside `SC2` executes on CPU 1.

When DEmOS is invoked with this configuration, processes `SC1-1` and `SC2-1` are started together at the beginning of the time window. `SC1-1` runs for 100 milliseconds; then, it is suspended and `SC1-2` is started, while `SC2-1` is still running. After `SC1-2` budget is up, CPU 0 stays idle until all safety-critical (SC) partitions are finished, and then `BE1-1` inside the `BE1` partition is started and runs until the end of the window. Then, the cycle repeats, as there is only a single window defined.

3.5 Partitions and processes

In this section, the design of partitions and processes is described, together with the dynamic process scheduler.

DEmOS partitions are groups of Linux processes, which are isolated from other partitions temporally (windows) and spatially (CPUs). Except for hardware-related side effects arising from shared CPU and memory resources, partitions do not influence one another. Partitions are implemented internally inside DEmOS and, except for the *cgroup* structure described below, not backed directly by any system features.

Existing Linux programs may be ran using DEmOS without any modification required. The scheduled processes are defined as shell commands (`cmd` key in the configuration file), which DEmOS executes during initialization using the `/bin/sh` shell. This provides flexibility in how the user chooses to start the tasks, and delegates argument parsing and similar issues to the native shell. The shell startup overhead is negligible for expected uses, as DEmOS is designed for long-running tasks.

Unlike in ARINC 653, where each process has a single thread of execution, multithreading in processes scheduled by DEmOS is supported. However, the threads are scheduled by the Linux kernel scheduler and not controlled by DEmOS. Multiprocessing (creating new child processes) is also supported, as it is needed for correct functionality of shell scripts and similar environments; all child processes are then treated as a single unit, scheduled together with the original parent process, with internal scheduling again done by the kernel scheduler.

3.5.1 Process scheduling

When a partition is running (scheduled inside the current time window), a second-level dynamic scheduler manages processes inside the partition. At a given time, at most a single process from the partition runs, across all CPUs available to the partition. Processes are scheduled in FIFO order.

Each process has a configured time budget (`budget` key in the configuration file, in milliseconds), which is replenished at the beginning of each window. When scheduled, the process runs either until the budget is exhausted, or until it signals completion [section 3.5.3]. The process selection algorithm of the scheduler differs based on whether the partition is safety-critical or best-effort.

Processes from a **safety-critical partition** are always scheduled starting from the first one, and must all complete before the end of the window. In a safety-critical system, failing to do so could result in a system failure; DEmOS outputs a warning when this occurs, but suspends the process and continues with execution of the next window.

In a **best-effort partition**, execution may reach the end of a window. Remaining budget of the current process is then preserved and in the next window where this partition is scheduled, execution continues from the point where it was previously stopped. For example, if a process with a 300 millisecond budget is started 100 milliseconds before the window ends, it will be preempted and then executed with a 200 millisecond budget next time the partition is scheduled.

Optionally, each process may also have a defined **budget jitter** (`jitter` key in the configuration file, also in milliseconds). This lets the user simulate processes with variable length of execution. If the jitter is non-zero, the actual budget is selected uniformly from the range $[budget - jitter/2,\ \ budget + jitter/2]$. The jitter must not be greater than 2 times the budget.

3.5.2 *cgroup* usage

DEmOS needs to control the processes by suspending/resuming and reacting to process exit. CPU affinity of the partition also needs to be changed, depending on the slice the partition is currently running under. To achieve these, the Linux *cgroup* feature is used — a cgroup is created for each partition, with another nested cgroup for each process in the partition.

When a partition is started, the `cpuset` cgroup is used to configure the allowed CPU set for the whole partition. The freezer cgroup is used to suspend and resume individual processes. As the child processes spawned by the original process may outlive their parent, the cgroup event monitoring functionality is used to detect when the cgroup is empty.

DEmOS needs to run under the `root` account to create the necessary cgroups automatically. Otherwise, DEmOS fails due to insufficient permissions and provides a list of shell commands to the user to create the cgroups manually.

3.5.3 Process client library

DEmOS provides a client library that scheduled processes may use to communicate and cooperate with the scheduler. Two functions are exposed: process initialization and yielding. To use the library, the program must be linked against it and explicitly call the provided API.

Process initialization

Many processes require a non-trivial amount of time to initialize to a working state. As the scheduling intervals may be quite short, it is better to run the initialization before the static scheduler starts. If the `init` boolean key in the configuration file is set, the process is scheduled outside the static windows and allowed to run until it signals that it completed the initialization by calling an appropriate API function.

To allow for correct detection of the available CPU count (e.g. for spawning worker threads), the initialization is done in the widest CPU set (the highest number of CPUs) the process will run on. All initialized processes are started in parallel and scheduled by the kernel scheduler — no isolation guarantees are provided during initialization.

Process completion (yielding)

In cases where the process finishes the required work for the current window before its budget is used up, it may yield the remaining CPU time through the library API. This will immediately suspend the process and schedule the next available one. The functionality is similar to the `sched_yield` POSIX system call, but applies to the whole process (and all child processes), not only the calling thread.

3.6 Power policies

One possible source of inconsistencies between successive runs of the same schedule is the kernel power management subsystem, including CPU frequency scaling and sleep states. At the same time, thanks to the static partition schedule, DEmOS has more information about the future CPU load distribution than the kernel, allowing it, at least in theory, to make better power management decisions in sync with the schedule, lowering power consumption and platform temperature while still providing maximum performance when required.

DEmOS implements a high-level interface to the *CPUFreq* kernel subsystem, allowing it to disable automatic kernel CPU frequency scaling and implement custom **power policies**, which observe various scheduler events and manage CPU frequencies accordingly. Multiple power policies are implemented, selectable using the `-p` command-line parameter. A simple interface to add custom power policies by modifying the DEmOS source code is provided, described further in the corresponding implementation section [section 4.8].

To access the kernel power management features, DEmOS must be running under the `root` account, or another access control mechanism must be used to provide write access to the *CPUFreq* `sysfs` interface.

3.7 Runtime output

As DEmOS is a CLI-only application, a status log is provided on the standard error output to inform the user of the current state. The granularity of the output can be configured by the user, and enough information is provided so that the user does not have to use external tools to understand the behavior of DEmOS and the scheduled processes.

4 Implementation

This section describes the internal implementation of DEmOS, including project structure, optimization and tooling. This may be of interest to future contributors and inquisitive end users.

DEmOS is implemented a single user-space C++17 Linux program called `demos-sched`, running with real-time priority [18]. An event-driven architecture is used, utilizing the *libev* library [19], which internally uses `epoll` to receive events. Linux *cgroup v2* [section 2.5] is used to control the scheduled processes. CPU frequency scaling is controlled using the `sysfs`-based *CPUFreq* interface [section 2.7.1].

Multiple instances of DEmOS may run in parallel, but at most one instance may have active power management enabled. As long as each instance uses different CPUs, there should be minimal interference between the instances, except for side effects like platform temperature, memory bus contention and similar.

4.1 Building

DEmOS uses the Meson build system [20] for building and running tests. A `Makefile` wrapper provides common configurations exposed as targets. Some of the more useful targets are:

- `make release`, which configures the project for a release build,
- `make debug`, which configures the project for a debug build (enables verbose trace messages, debugger symbols and address sanitization),
- `make aarch64`, which configures the project for cross-compilation to 64-bit ARM architecture,
- `make`, which builds the project using current configuration,
- `make test`, which builds the project and then runs the test suite [section 4.11].

External dependencies (*libev*, *yaml-cpp* and *spdlog*) may either be installed using the system package manager, or cloned as `git` submodules. In both cases, the dependencies are statically linked.

For native builds, the resulting `demos-sched` binary is written to `build/src/demos-sched`. For ARM cross-compiled builds, it is written to `build-aarch64/src/demos-sched`.

4.2 Event loop

After a synchronous initialization, DEmOS operates by reacting to timers, messages from processes and system signals. Therefore, an event-driven architecture was chosen, using the C++ interface of the C-based *libev* library (`ev++.h`). Custom wrappers were added for *eventfd* and *timerfd*, sub-classing the `ev::io` *libev* class. File writes to the *cgroup* and *sysfs* virtual files are done synchronously, as the written data buffers are too small to justify the additional overhead of asynchronous writes.

4.3 Project and module structure

The project directory structure is as follows:

- The `src` directory contains the source code of DEmOS.
- The `lib` directory contains the process library sources. [section 3.5.3]
- The `test` directory contains automated tests, implemented as a set of `bash` scripts. [section 4.11]
- The `src/tests` directory contains a set of testing processes, used by the automated tests.
- The `test_config` directory contains DEmOS configuration files that demonstrate some of the supported features.
- The `subprojects` directory contains external libraries used by DEmOS, managed as `git` submodules.

Header and implementation files are always in the same directory, using the same base-name — therefore, there is a one-to-one correspondence between a header file and its implementation. Some classes are implemented as a header-only module, typically for classes which are only used in a single compilation unit. C++ classes are used to implement most of the functionality, but inheritance, polymorphism and other patterns common for object-oriented programming are avoided in most of the codebase.

Main entry point of the scheduler is the `main.cpp` file, which parses the command-line arguments, configures the selected power policy (if any), and invokes the `Config` module to parse the configuration file and create the scheduler objects (`MajorFrame`, `Window`, `Slice`, `Partition`, `Process`). Then, an instance of the `DemosScheduler` class is created, which starts the event loop, runs the process initialization using the `PartitionManager` class and then starts and manages the scheduler. The `Cgroup` module is used internally by the `Partition` and `Process` classes to manage the associated cgroups.

4.4 *cgroup* interface

Currently, DEmOS requires the *cgroup* hierarchy to be mounted in hybrid mode [section 2.5.1]. However, the cgroup tree structure created during initialization is the same for each controller. Therefore, it should be possible to add support for unified mode in the future without major modifications. Together with `systemd-run` [21], this would allow DEmOS to run under non-`root` accounts without manual *cgroup* creation.

During initialization, handled by the `cgroup_setup` module, the `/proc/self/cgroup` virtual file is parsed to retrieve the cgroups that DEmOS is running under, and the child cgroup used by DEmOS is then created under the corresponding parent cgroup. This way, a user may limit resources used by scheduled processes by assigning DEmOS to a custom cgroup.

Each DEmOS instance configures a top-level cgroup called `demos-<pid>`, where `<pid>` is the process ID of the instance. This allows multiple instances to run in parallel. When a `Partition` is instantiated, it creates a child cgroup with the same name. When a `Process` is added to the partition, a child cgroup is also created, using a sequentially generated name. The resulting cgroup structure is `.../demos-<pid>/<partition_name>/proc<process_index>`. The same structure is mirrored between the `cpuset`, `freezer` and `unified` hierarchies used by DEmOS.

The `Cgroup` class is used to encapsulate manipulation of the *cgroup* virtual files, with child classes for each type of *cgroup* controller. The corresponding cgroup is automatically created when the class is instantiated and removed on instance destruction (the RAII pattern).

If DEmOS crashes or freezes during execution, the created cgroups will not be cleaned up. In this case, a shell script can be found at `src/cleanup_crash.sh`, which stops all running DEmOS instances and cleans up any remaining cgroups.

4.5 Process client library

The process library is implemented as a C99 static library. A Meson target is provided to allow simple linkage to the library. *eventfd* file descriptors are used for communication with DEmOS.

The library exposes three functions, all of which return zero on success and a negative integer when an error occurs:

- `int demos_init()` — Initializes the library by loading parameters from the `DEMOS_PARAMETERS` environment variable. If not called explicitly, it is automatically invoked the first time any other function from the library is called.

- `int demos_initialization_completed()` — Informs DEmOS that the process completed its initialization, and immediately suspends the process. If the process was not configured to initialize in the configuration file (`init: yes`), this function immediately returns an error status and lets the program continue execution.
- `int demos_completed()` — Yields the remaining CPU time for the current window, immediately suspending the process.

```c
#include "demos-sch.h"
#include <err.h>

int main()
{
    // initialize the library, checking for errors
    if (demos_init() < 0) {
        err(1, "demos_init");
    }

    // initialize the program here

    // signal that initialization is completed
    if (demos_initialization_completed() < 0) {
        err(1, "demos_initialization_completed");
    }

    while (1) {
        // do work for current window, or exit if done

        // yield the remaining budget for this window (signal completion)
        if (demos_completed() < 0) {
            err(1, "demos_completed");
        }
    }
}
```

Fig. 6. A code example demonstrating the process library usage.

4.6 Scheduler objects

This section describes the high-level implementation of scheduler objects mentioned in the Design section of this thesis.

4.6.1 `Process`

The `Process` class is implemented as a passive interface to the underlying system process — it accepts commands from other DEmOS components and forwards events received from the system processes without any active processing.

The `suspend()` and `resume()` methods freeze and thaw the corresponding freezer cgroup, respectively. The `exec()` method spawns the underlying system process and sets up a pair of *eventfd* file descriptors, which are used by the process client library [section 3.5.3] to signal completion. To do so, the library writes to the first descriptor, and then initiates a blocking read on the second descriptor, which prevents further execution of the process. A completion callback method on the corresponding `Process` instance is invoked as a result of the write, which propagates the event to the parent `Partition`. Next time the process is resumed, `Process` writes to the second file descriptor, which ends the blocking read in the process library and allows it to continue execution.

When all processes in the cgroup (the original spawned process and all child processes) terminate, the event is also propagated to the parent `Partition`.

4.6.2 `Partition`

The `Partition` class is also a passive interface to the contained `Process` instances. Of interest is the `reset()` and `disconnect()` method pair, which is used both during process initialization and during normal scheduling windows to attach/detach to the `Partition` instance. `reset()` is called with the desired CPU set the processes should run on, and a process completion callback. Until `disconnect()` is called, all process completions events invoke the provided callback.

`PartitionManager`

The `PartitionManager` class contains all defined `Partition` instances, coordinates the process initialization and listens for process termination events. When all processes terminate, it notifies the main `DemosScheduler` instance, which then stops the scheduler.

4.6.3 `MajorFrame`

The `MajorFrame` class implements the static partition scheduler. It reuses a single *timerfd* instance to periodically schedule the time windows by indirectly starting and stopping the `Slice` instances contained in each `Window` instance.

4.6.4 `Slice`

The `Slice` class implements the process scheduler. When the parent `Window` is started, it attaches to the scheduled partitions, asynchronously iterates over all pending processes and resumes them one by one, letting each execute until either the time budget runs out or the process signals completion.

4.7 Synchronization messages

DEmOS may be ran by another program which needs to synchronize with DEmOS time windows. One such example is the thermobench [22] tool, which measures the thermal properties of the SoC the scheduled processes are running on. To facilitate this, DEmOS provides the `-m <message>` and `-M <message>` command-line parameters, which cause DEmOS to print the provided message to the standard output at the beginning of each window or each major frame, respectively.

4.8 Power management

The power management module needs to receive events from multiple parts of the scheduler. The abstract class `SchedulerEvents` specifies hooks (event handlers) for events relevant for the power management (window switch, partition completion, etc.). An instance of this class is passed to all relevant scheduler objects, which directly call the event handlers when relevant events occur. This approach was chosen over a more generic event propagation architecture to keep the code paths short and easily auditable, as most of the event handlers are called from latency-sensitive code paths.

The `SchedulerEvents` class is implemented by a number of so-called power policies, which use the `PowerManager` module described below to change CPU P-states in reaction to the received events. One such power policy is the `MinBE`, which executes safety-critical partitions in the highest P-state and best-effort partitions in the lowest one:

```
class PowerPolicy_MinBE : public PowerPolicy {
private:
    PowerManager pm{};
public:
    PowerPolicy_MinBE() {
        // run initialization on max frequency
        for (auto &p : pm.policy_iter()) p.write_frequency(p.max_frequency);
    }
    void on_sc_start(Window &) override {
        // run SC partitions on max frequency
        for (auto &p : pm.policy_iter()) p.write_frequency(p.max_frequency);
    }
    void on_be_start(Window &) override {
        // run BE partitions on min frequency
        for (auto &p : pm.policy_iter()) p.write_frequency(p.min_frequency);
    }
};
```

Fig. 7. Code sample of the `MinBE` power policy, one of the predefined power policies available in DEmOS.

4.8.1 `PowerManager` and `CpufreqPolicy`

The `PowerManager` class wraps the user-space *CPUFreq* interface. During initialization, it configures the *CPUFreq* driver, switching the *intel_pstate* driver to the passive mode, if applicable. Then, an instance of the `CpufreqPolicy` class is created for each existing *CPUFreq policy* kernel object, which automatically activates the `userspace` governor The instance provides a list of available frequencies, which may be set using the `write_frequency()` method — the frequency is applied for all CPUs managed by the policy.

4.9 Runtime logging

A single global instance of the *spdlog* library [23] logger is used for logging to the standard error output. The user selects the desired log level by setting the `SPDLOG_LEVEL` environment variable; the `info` log level is selected by default.

Trace messages are logged for all operations using the `SPDLOG_LOGGER_TRACE()` macro, which can be deactivated at compile-time to reduce overhead for the release build. It is possible to fully reconstruct the scheduler run from the trace logs.

The warning and error messages are designed to communicate the issue clearly and offer solution and workarounds. As an example, the following warning message is emitted when a potential conflict with another program is detected during *CPUFreq policy* initialization:

```
>>> 10:23:09.762 [warning] `cpufreq` governor is already set to 'userspace'.
    This typically means that another program (or another instance of DEmOS)
    is already actively managing CPU frequency scaling from userspace. This DEmOS
    instance was started with power management enabled, and it will overwrite the
    CPU frequencies the other program may have set up.

    Note that running multiple DEmOS instances with an active power policy
    is NOT supported, and later instances may crash when the first one exits.
```

Fig. 8. An example of a warning message outputted by DEmOS.

4.10 Performance considerations

Although DEmOS is not a hard real-time application, low predictable latency and minimal overhead are still important. Several steps were taken during development to ensure good performance.

First, all scheduler objects are allocated during initialization, and no dynamic heap allocation is allowed after the scheduler starts, to avoid unpredictable allocator latency. Stack-based buffers of fixed size are used for I/O. To verify, an allocation tracker module was developed, which logs all allocations occurring after initialization. Internally, it overrides the `new` and `delete` C++ operators. While this does not capture direct calls to `malloc()` (which is not used in DEmOS) and internal *libc* allocations, it still exposes many subtle C++ issues such as accidental object copies. The allocation tracker is enabled in a separate build configuration invoked by `make alloc_test`.

Second, `fstream`-based I/O is only allowed in non-critical sections of code. For latency-sensitive cgroup manipulation and similar I/O, cached raw file descriptors are used with the raw `write` and `read` POSIX system calls. This avoids internal buffering and object creation overhead.

Third, indirection and abstraction is minimized throughout the codebase. The interfaces are designed to be hard to misuse and well-documented, but the control flow and full code path must be always clearly visible. Specifically, dynamic dispatch and generic event chains are avoided where possible, preferring direct method calls, with code paths going through the minimum necessary number of modules. For critical scheduler functionalities, the code paths were analyzed line by line to ensure no unpredictable behavior.

4.11 Automated testing

An automated test suite to validate correctness of DEmOS functionality is implemented, and may be ran through Meson using `make test`. The tests are written as `bash` scripts, using the `tap-functions` library [24], which outputs the test results in the Test Anything Protocol (TAP) [25], which is supported by Meson. All test scenarios

invoke the `demos-sched` command externally; the internal behavior is currently not tested.

4.12 Software compatibility

DEmOS works on all Linux distributions where *cgroup v2* is supported and mounted in hybrid mode, including the Microsoft WSL2 kernel, which was used for development. For the optional power management, the *CPUFreq* kernel subsystem is required.

5 Evaluation

This section describes the platform DEmOS was developed and tested on and presents benchmarks of the functionality documented in previous sections.

5.1 Evaluation platform

For evaluation of thermal characteristics, the testbed described in a previous paper [25] was used, with the ARM64-based Toradex i.MX8QuadMax platform. On software side, the Linux kernel version 5.4.70 from the official Yocto distribution provided by Toradex was used, with a custom Debian distribution providing the user-space environment. For measuring thermal properties, the Thermobench tool was used.

The option of compiling a custom Linux kernel was explored, but ensuring full compatibility, especially with regards to the required GPU drivers proved problematic, and as the required power management features could be implemented without modifications to the kernel, the official kernel image was used instead.

5.2 Basic functionality

All features described in this thesis are implemented, and most are tested using the automated test suite. Test configurations are prepared for features which are hard to test in an automated way.

5.3 Stress testing

To ensure that DEmOS scales well for a larger number of scheduler objects, multiple stress tests were done. To run the larger configurations, the limit on the number of opened file descriptors had to be manually raised by calling `ulimit -n 100000`, as DEmOS keeps a file descriptor opened for each cgroup it manages, and two file descriptors for each managed process.

First, a configuration of 10 000 partitions was evaluated, each containing one process. Due to the way DEmOS spawns the scheduled processes, where each forked child process is immediately suspended, in combination with higher memory usage by DEmOS itself, the system ran out of memory during process initialization and invoked the OOM killer mechanism. Interestingly, even the copy-on-write mechanism used by Linux [26] to lower the physical memory usage of forked processes did not work well enough to allow DEmOS to finish initialization.

For a smaller set of 1 000 partitions, each containing one process, DEmOS successfully initialized all processes and started scheduling in 9.3 seconds, on average over 8 runs. The initialization is dominated by the time to spawn all scheduled processes (6.1 seconds on average). After initialization, the scheduler is responsive and its scheduling latency (see below) is not dependent on the number or scheduled partitions.

Next, a set of 1200 processes, split equally between 3 partitions was tested. DEmOS also successfully initialized in 4.0 seconds on average over 8 runs, with process spawning again taking up majority of the initialization time.

5.4 Evaluation of power management

To validate that the implemented power management feature lowers the operating temperature as expected, the ADASMark™ [2] automotive benchmark suite was used. For all tests, 400 frames were processed in each benchmark run, and the final temperature of the CPU package was recorded.

The benchmark was ran 5 times outside of DEmOS, using all 6 available CPUs in highest P-state (1.5 GHz and 1.2 GHz for the A72 and A53 cores, respectively). Then, another 5 runs were recorded under DEmOS, with a power policy that set the minimal supported frequency (600 MHz) for all cores.

Without DEmOS, the average recorded temperature from all runs was 61.6 °C. With the power management applied, the averaged temperature dropped to 53.3 °C. The idle temperature of the board was recorded at 44.6 °C.

5.5 Scheduling overhead

To test the scheduling overhead, the same 400 frame ADASMark™ benchmark was used. Each of the following measurements was repeated 5 times and averaged. First, the benchmark was ran outside of DEmOS to get a baseline; here, each frame was processed in 404 milliseconds on average. Then, the benchmark was ran under DEmOS in a single repeating window, with windows lengths of 1000, 100, 10, 5, 2 and 1 milliseconds. The results are plotted in Fig. 4.

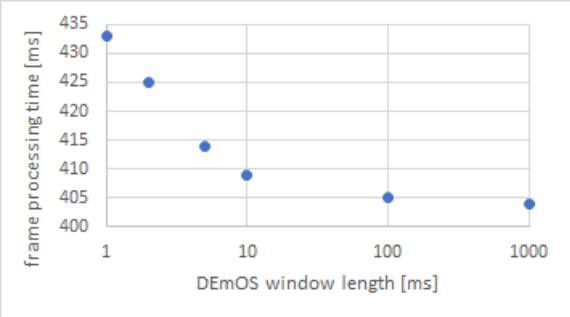

Fig. 9. Average time for processing of one frame for varying window lengths.

By subtracting the baseline frame time and dividing the difference by the number of DEmOS major frames that were executed per single frame, we get an estimate of the overhead DEmOS adds for each window switch. For the 1 millisecond window length, the estimated overhead is approximately 60 microseconds.

6 Conclusion

As a result of this project, DEmOS, a Linux user-space scheduler was released. Compared to the original prototype, it is more robust and maintainable, better documented, and offers multiple new features, most important of which is the power management system.

As shown by the evaluation and our previous paper [4], although DEmOS is not a truly real-time scheduler, its overhead is low enough for further experimental work.

6.1 Possible improvements

- As shown by the stress testing, the approach used for spawning the scheduled processes is non-optimal and replacing the `fork` and `exec` with `posix_spawn` could improve the initialization time and scalability.
- In addition to the *CPUFreq* subsystem already used by DEmOS, Linux kernel provides the *CPUIdle* subsystem, which manages CPU sleep states. Thanks to the static partition schedule, DEmOS has precise information about the available idle time, which should in theory allow it to lower the thermal output by controlling the CPU sleep states without affecting performance.
- To aid analysis of schedule execution, DEmOS could store machine-readable logs during operation, which could be later use to reconstruct and visualize the schedule directly, without using external tools.

7 References

[1] *Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations* (2005), RTCA DO-297 / EUROCAE ED-124

[2] EMBC, *The ADASMark™ Benchmark* (2018), [Online]. Available: https://www.eembc.org/adasmark/ (Viewed 2021-05-21)

[3] Ellis, M., Anderson, W., and Montgomery, J., *Passive Thermal Management for Avionics in High Temperature Environments* (2014), SAE Technical Paper 2014-01-2190, 2014, [Online]. Available: https://www.sae.org/publications/technical-papers/content/2014-01-2190/

[4] Ondřej Benedikt, Michal Sojka, Pavel Zaykov, David Hornof, Matěj Kafka, Přemysl Šůcha, Zdeněk Hanzálek, *Thermal-aware scheduling for MPSoC in the avionics domain: Tooling and initial results* (in submission) (2021)

[5] *DEmOS scheduler* (2021), [Online]. Available: https://github.com/CTU-IIG/demos-sched (Viewed 2021-05-21)

[6] Jane W. S. Liu, *Real-Time Systems* (2000), U.s.a: Prentice Hall, June 15, 2000, ISBN 9780130996510.

[7] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni, *A Survey on Cache Management Mechanisms for Real-Time Embedded Systems.* (2015), ACM Comput. Surv. 48, 2, Article 32 (November 2015).

[8] *CFS: Completely fair process scheduling in Linux* (2019), [Online]. Available: https://opensource.com/article/19/2/fair-scheduling-linux (Viewed 2021-05-21)

[9] *The real-time endgame is moving quickly now* (2020), [Online]. Available: https://wiki.linuxfoundation.org/realtime/rtl/blog (Viewed 2021-05-21)

[10] *"Department of Defense - Standard Practice Safety System"* (2012), MIL-STD-882E.

[11] ARINC, *"Avionics Application Software Standard Interface Part 1 - Required services (ARINC Specification 653P1-2)"* (2019), Airlines Electronic Engineering Committee - Aeronautical Radio, Inc. (ARINC), 2551 RivaRoad, Annapolis, Maryland 21401-7435, Tech. Rep.

[12] S. H. VanderLeest, *"ARINC 653 hypervisor,"* (2010), 29th Digital Avionics Systems Conference, 2010, pp. 5.E.2-1-5.E.2-20.

[13] DDC-I, *Deos, a Time & Space Partitioned, Multi-core Enabled, DO-178C DAL A Certifiable RTOS* (2021), [Online]. Available: https://www.ddci.com/products_deos_do_178c_arinc_653/ (Viewed 2021-05-21)

[14] *Control Group v2* (2015), [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v2.txt (Viewed 2021-05-21)

[15] systemd, *Control Group APIs and Delegation* (2018), [Online]. Available: https://systemd.io/CGROUP_DELEGATION/ (Viewed 2021-05-21)

[16] *CPU Performance Scaling* (2017), [Online]. Available: https://www.kernel.org/doc/html/v5.12/admin-guide/pm/cpufreq.html (Viewed 2021-05-21)

[17] *intel_pstate CPU Performance Scaling Driver* (2017), [Online]. Available: https://www.kernel.org/doc/html/v5.12/admin-guide/pm/intel_pstate.html (Viewed 2021-05-21)

[18] *sched (7) — Linux manual page*, [Online]. Available: https://man7.org/linux/man-pages/man7/sched.7.html (Viewed 2021-05-21)

[19] *libev*, [Online]. Available: http://software.schmorp.de/pkg/libev.html (Viewed 2021-05-21)

[20] *The Meson Build system*, [Online]. Available: https://mesonbuild.com/ (Viewed 2021-05-21)

[21] systemd, *systemd-run*, [Online]. Available: https://www.freedesktop.org/software/systemd/man/systemd-run.html (Viewed 2021-05-21)

[22] *Thermobench* (2021), [Online]. Available: https://github.com/CTU-IIG/thermobench (Viewed 2021-05-21)

[23] *spdlog* (2021), [Online]. Available: https://github.com/gabime/spdlog (Viewed 2021-05-21)

[24] *TAP functions for bash* (2012), [Online]. Available: https://github.com/goozbach/bash-tap-functions (Viewed 2021-05-21)

[25] *Test Anything Protocol* (2021), [Online]. Available: https://testanything.org/ (Viewed 2021-05-21)

[26] Michal Sojka, Ondřej Benedikt, Zdeněk Hanzálek, *Testbed for thermal and performance analysis in MPSoC systems* (2020), DOI: 10.15439/2020F174, [Online]. Available: https://annals-csis.org/proceedings/2020/drp/pdf/174.pdf

[27] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe, *A fork() in the road* (2019). In Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy. ACM, New York, NY, USA. https://doi.org/10.1145/3317550.3321435

Appendix 1: DEmOS command-line interface

The following command-line parameters are supported:

- `-c <CONFIG_FILE>` — Path to the configuration file.
- `-C <CONFIG_STRING>` — Inline configuration string in YAML format.
- `-p <POWER_POLICY>` — Name of the selected power management policy. If not set, power management is disabled. If multiple instances of DEmOS are running in parallel, this parameter must not be passed to more than one instance.
- `-g <CGROUP_NAME>` — Name of the root cgroups created by DEmOS. If not set, `demos-<pid>` is used, where `<pid>` is the process ID of the current instance.
- `-m <WINDOW_MESSAGE>` — If set, DEmOS prints `WINDOW_MESSAGE` to the standard output at the beginning of each time window.
- `-M <MF_MESSAGE>` — If set, DEmOS prints `MF_MESSAGE` to the standard output at the beginning of each major frame.

Appendix 2: DEmOS configuration file format

Configuration files are written in the YAML format. They can have either canonical or simplified form, with the latter being automatically converted to the former. Canonical form is most flexible, but for same cases a bit verbose. Verbosity can be reduced by using the simplified form. Both forms are described bellow.

Canonical form of the configuration file

Configuration file is a mapping with the `set_cwd`, `partitions` and `windows` keys:

- `set_cwd` (optional, default: true) is a boolean specifying whether all scheduled processes should have their working directory set to the directory of the configuration file; this allows you to safely use relative paths inside the configuration file and scheduled programs.
- `partitions` is an array of partition definitions.
    - *Partition definition* is a mapping with `name` and `processes` keys.
    - `processes` is an array of process definitions.
    - *Process definition* is mapping with `cmd`, `budget`, `jitter` and `init` keys.
        - `cmd` is a string with a command to be executed (passed to `/bin/sh -c`).
        - `budget` specifies process budget in milliseconds.
        - `jitter` (optional, default: 0) specifies jitter that is applied to the budget for each invocation.
        - `init` (optional, default: false) is a boolean specifying if process should be allowed to initialize before scheduler starts

- `windows` is an array of window definitions.
    - *Window definition* is a mapping with `length` and `slices` keys.
        - `length` defined length of the window in milliseconds.
        - `slices` is an array of slice definitions.
    - *Slice definition* is a mapping with the `cpu` key and optional `sc_partition` and `be_partition` keys.
        - `cpu` is a string defining scheduling CPU constraints. The value can specify a single CPU by its zero-based number (e.g. `cpu: 1`), or a range of CPUs (`cpu: 0-2`), or combination of both (`cpu: 0,2,5-7`).
        - `sc_partition` and `be_partition` are strings referring to partition definitions by their `name`s.

Example canonical configuration can look like this:

```
set_cwd: yes

partitions:
  - name: SC1
    processes:
      - cmd: ./safety_critical_application
        budget: 300
        init: yes
  - name: BE1
    processes:
      - cmd: ./best_effort_application1
        budget: 200
  - name: BE2
    processes:
      - cmd: ./best_effort_application2
        budget: 200
      - cmd: ./best_effort_application3
        budget: 200

windows:
  - length: 500
    slices:
      - cpu: 1
        sc_partition: SC1
        be_partition: BE1
      - cpu: 2,5-7
        be_partition: BE2
```

Simplified form of the configuration file

- The `slice` keyword may be omitted. Then it is expected that there is just one slice inside window scheduled at all CPUs.

```
{
  partitions: [ {name: SC, processes: [{cmd: echo, budget:100}]
}],
  windows: [ {length: 500, sc_partition: SC} ]
}
```

  is the same as

```
partitions:
  - name: SC
    processes:
      - cmd: echo
        budget: 300
windows:
  - length: 500
    slices:
      - cpu: 0-7
        sc_partition: SC
```

- Partition may be defined directly inside `windows`.

```
windows: [ {length: 500, sc_partition: [{cmd: proc1, budget:
500}] } ]
```

  is the same as

```
partitions:
  - name: anonymous_0
    processes:
      - cmd: proc1
        budget: 500
windows:
  - length: 500
    slices:
      - cpu: 0-7
        sc_partition: anonymous_0
```

- If process `budget` is not set, then the default budget `0.6 * length` of window is set for `sc_partition` processes and `length` of window is set for `be_partition` processes.

- You can use `xx_processes` keyword for definition of partition by the list of commands:

```
windows: [ {length: 500, sc_processes: [proc1, proc2]} ]
```