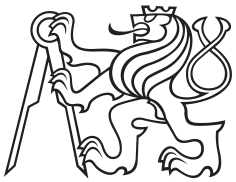
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Proactive and reactive approaches for non-critical tasks scheduling under thermal constraints in the avionics domain

Radek Bumbálek

Supervisor: Ing. Ondřej Benedikt
Field of study: Open Informatics
Subfield: Software Engineering
January 2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bumbálek** Jméno: **Radek** Osobní číslo: **459128**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Proaktivní a reaktivní přístupy pro rozvrhování nekritických úloh respektující tepelná omezení v avionické doméně

Název diplomové práce anglicky:

Proactive and reactive approaches for non-critical tasks scheduling under thermal constraints in the avionics domain

Pokyny pro vypracování:

With increasing demands for the high performance of embedded systems, thermal management has become an inseparable part of the system's design.
One of the techniques used in thermal management is task scheduling.
It is well known that there exists a trade-off between the system's performance and temperature. Consequently, new scheduling methods are being developed (i) to maximize the system's performance while respecting the given thermal envelope, or (ii) to minimize the system's temperature while guaranteeing that the critical tasks will be completed in time. The performance and complexity of the methods vary, depending on the amount of information they use; for example, by fully analyzing the behavior of the tasks, one can create very complex models (with very limited scalability) and pre-compute the schedule offline. On the other hand, one can react to the changes of the system's state by following relatively simple rules.
The goal of this thesis is to evaluate different techniques used for task scheduling under thermal constraints. The evaluation will be done using real hardware. Also, the student should assess which of the techniques are suitable for scheduling the non-critical workloads in the avionics domain, where the tasks are executed within time windows, securing their temporal isolation.
Student will:
- Review the existing works on periodic tasks scheduling under thermal constraints.
- Select suitable methods for use on real hardware (consider, for example, GRUB-PA [1], and MultiPAWS provided by the supervisor), and evaluate the trade-offs between their complexity (addressing both – implementation complexity and computational complexity) and performance.
- Propose a new method for periodic task scheduling under thermal constraints assuming the time windows imposed by an application in the avionics domain.
- Implement the proposed method, design testing scenarios, and prepare benchmark instances for testing.
- Experimentally evaluate the proposed method on real hardware (e.g., i.MX 8QuadMax), and compare it with existing methods.

Seznam doporučené literatury:

[1] Scordino, Claudio, et al. "Energy-Aware Real-Time Scheduling in the Linux Kernel." Proceedings of the 33rd Annual ACM Symposium on Applied Computing, ACM, 2018, pp. 601–08. DOI.org (Crossref), doi:10.1145/3167132.3167198.
[2] Mascitti, Agostino, et al. "An Adaptive, Utilization-Based Approach to Schedule Real-Time Tasks for ARM Big.LITTLE Architectures." ACM SIGBED Review, vol. 17, no. 1, July 2020, pp. 18–23. DOI.org (Crossref), doi:10.1145/3412821.3412824.
[3] Hanumaiah, V., et al. "Performance Optimal Online DVFS and Task Migration Techniques for Thermally Constrained Multi-Core Processors." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 11, Nov. 2011, pp. 1677–90. DOI.org (Crossref), doi:10.1109/TCAD.2011.2161308.
[4] Zhou, Junlong, et al. "Thermal-Aware Task Scheduling for Energy Minimization in Heterogeneous Real-Time MPSoC Systems." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 8, Aug. 2016, pp. 1269–82. DOI.org (Crossref), doi:10.1109/TCAD.2015.2501286.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Ondřej Benedikt, katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **24.06.2021** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **19.02.2023**

Ing. Ondřej Benedikt podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ _____

Datum převzetí zadání Podpis studenta

Acknowledgements

Primarily I would like to thank my supervisor Ing. Ondřej Benedikt for his guidance and willingness to help, especially in theoretical aspects of my work. Because of COVID pandemics, most of my work was done over the distance, which was only possible thanks to Michal Sojka PhD., who guided my work on remote devices.

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 instruction for observing the ethical principles in the preparation of university thesis.

V Praze, 4. January 2022

Abstract

The goal of this diploma thesis is to use best-effort task scheduling to lower the temperature. To achieve this goal, the problem is formulated as a linear program. Our formalization respects characteristics of tasks and heterogeneous architecture of target multiprocessor system on chip (MPSoC). The main tool to achieve a reduction of produced heat is dynamic voltage and frequency scaling (DVFS). The scheduler is evaluated on real hardware and measured results are put in comparison to state-of-the-art Linux scheduler SCHED_DEADLINE. Moreover, reactive techniques are proposed, to avoid possible overheating caused by a change of the environmental conditions or imprecisions of the model at runtime.

Keywords: scheduling, MPSoC, DVFS, linear programming, proactive scheduling, offline scheduling, reactive policy, hardware experiments, best-effort task scheduling

Supervisor: Ing. Ondřej Benedikt

Abstrakt

Cílem této diplomové práce je vytvořit rozvrhovač best-effort úloh, který sníží teplotu zařízení při stejném množství vykonané práce. Abychom toho dosáhli, formulujeme problém pomocí lineárního programování. Naše reprezentace respektuje vlastnosti heterogenní architektury vybraného procesoru i vlastnosti úloh samotných. Hlavním nástrojem použitým pro docílení teplotní redukce je dynamické škálování frekvence. Rozvrhy vytvořené rozvrhovačem jsou otestovány na reálném zařízení a porovnány s výsledky systémového rozvrhovače SCHED_DEADLINE. Součástí práce jsou i návrhy reaktivních přístupů, které využívají již zmíněné rozvrhy a jsou schopné reagovat na změny okolních podmínek.

Klíčová slova: rozvrhování, MPSoC, DVFS, lineární programování, proaktivní rozvrhování, offline rozvrhování, rozvrhování best-effort úloh, reaktivní přístupy, experimenty na reálném zařízení

Překlad názvu: Proaktivní a reaktivní přístupy pro rozvrhování nekritických úloh respektující tepelná omezení v avionické doméně

Contents

Figures

Tables

Part I

Theory

Chapter 1

Introduction

Modern systems grow in complexity every year due to demand for higher performance. This phenomenon leads to the employment of Multiprocessor System on Chip (MPSoC), where often heterogeneous architecture allows for higher efficiency and computational throughput. However, the high performance of embedded MPSoC leads to heating the hardware. While active cooling offers a great potential to keep the temperature on a stable level without introducing additional needs for hardware or software changes, in some cases an additional device may not be desirable. Its size and additional energy consumption reduce its viability in the field of portable devices, whereas in safety-critical domains such as aerospace, any additional device might increase the possibility of a mechanical failure, especially under harsh conditions tied up with this field.

Software solutions are not burdened with these disadvantages. Furthermore, multiple techniques can be introduced. In general, we distinguish between proactive (sometimes also referred to as offline) and reactive (online) approaches. A proactive approach strives to prevent an undesirable event to happen (i.e. a system overheating). Static scheduling belongs among proactive techniques. With prior knowledge of tasks, an elaborate schedule can be prepared, which would lower the produced heat with low or no impact on the task execution.

The other set of approaches is called reactive. A reactive technique is triggered at the moment when an undesirable event happens. Reactive techniques can be provided through a set of policies, which are to be used based on the imminent state of the device. These reactive techniques offer great flexibility in thermal management; however, it often leads to lowering the performance. Unlike scheduling, reactive policies are less burdened with assumptions and more reflect the real state of the device.

Even in safety-critical systems, not all tasks are rated as critical. There are also so-called best-effort tasks present. Scheduling these tasks allows for a less restrictive approach, while a successful solution improves the efficiency of

the whole system. Even though reactive approaches introduce performance reduction and lack of execution guarantee, which poses a serious drawback for critical tasks, they can be admissible for best-effort tasks.

In this thesis, we focus on best-effort tasks. Our goal is to design, implement and evaluate a best-effort task scheduler, which minimizes the heat produced and allows for better thermal management. We demonstrate the efficiency of our scheduler by comparison with Linux deadline scheduler (SCHED_DEADLINE) which is a standard part of Linux kernel since version 3.14.

Moreover, we implement several reactive policies, which can prevent system overheating in runtime. We compare reactive policies with precomputed schedules, where the workload is scaled down. We discuss these approaches as possibilities to keep the system running under thermal constraints.

In Chapter 2 we describe existing research in the field. Chapter 3 explains the problem we are solving in more detail. In Chapter 4 we describe our model and scheduling algorithm used for the scheduler in detail. Chapter 5 summarizes the environment used for experiments. This includes the hardware we are using and environment-specific constants which has to be measured in advance. Methodologies and results of these measurements are also presented here. Implementation of our scheduler, as well as our test instance generator and additional tools which made this work possible can be found in Chapter 6. Finally, Chapter 7 consists of experiments and their results. Conclusion and future work is presented in Chapter 8.

Chapter 2

Related work

There have been many works submitted in the field of CPU task scheduling since resource allocation is one of the key components of any operating system. Moreover, with new techniques in the semiconductor industry, smaller and more powerful hardware is being made. With higher frequencies and smaller sizes, power density scales up, which makes energy-aware scheduling even more relevant.

In Section 2.1 we discuss the usage of various tools for thermal management seen in other works. Because our work combines proactive and reactive approaches, we describe related work in these two fields in Section 2.2 and Section 2.3 respectively. Works of significant impact are summarized in Section 2.4.

2.1 Tools for thermal management

First, the most prevalent tool for thermal management is Dynamic Voltage and Frequency Scaling (DVFS). DVFS is nothing rare in modern hardware. Multiple thermal or energy-aware scheduling approaches take advantage of it ([7], [10], [12], [13], [14], [15], [16], [17]).

There are often limits of real hardware, which are not considered in theoretical works. Frequency scaling is often not continuous as assumed in [7], but only certain discrete frequency levels are allowed. Also, it is not always possible to change frequency per core. In our case, we work with heterogeneous architecture, where only a per-cluster frequency setting is permitted.

The second tool is a task to core allocation or migration. In MPSoC, some migrating between cores or clusters may contribute to better thermal profile, due to spatial heat distribution, as shown in [6] and [7]. More commonly heterogeneous architecture is used. A detailed description of taking advantage of ARMs big.LITTLE architecture can be found in [18]. Similar approaches

can be found in [4], [8] and [5]. While we do not assume task migration due to strict cluster affinity of tasks in our model, we discuss the possible utilization of heterogeneous MPSoC in future work (Section 8.1).

For the sake of completeness, we also mention task reclamation as another technique, frequently used with best-effort task scheduling. Often task duration is described by Worst-Case Execution Time (WCET), however in case that execution time is shorter than WCET, the remaining time can be utilized to lower the temperature as shown in [1], [3], [15], [17] and [18].

2.2 Proactive approach

Proactive (offline) scheduling is not as strictly restricted in means of computational complexity as reactive (online) scheduling, where the schedule is being created at runtime. Therefore, it allows for the usage of more complex models. We describe these models with an algebraic modeling language. This representation can be solved via modern solvers and in general, it presents an optimization problem. The schedule is refined from its solution, as seen in [2], [4], [5], [8], [10] and [13].

Models are often created to allow for easy extraction of the final solution (e.g. in our case we utilize McNaughton's algorithm). Therefore, most works focus on solving the optimization problem itself. In [2] Artificial Neuron Networks are used to speed up the computation. With achieved speedup, authors there discuss possible usage in online scheduling. The same discussion can be seen in [13], where the speedup is achieved by approximation techniques.

Another option to avoid the high complexity of optimization is by relaxation of the integral constraints as shown in [5], where the Mehepu algorithm is presented. After solving a relaxed problem, the final task distribution is solved via multiple bin-packing approaches.

A different approach uses the mathematical model to derive an algorithm or a heuristic, solving the task allocation problem. Zhou et al. [16] derive a two-stage algorithm, where energy-aware task allocation is performed in the first stage, while the second stage focuses on lowering peak energy. The final solution is then enhanced by a slack distribution which effectively lower temperature peaks.

2.3 Reactive policies

Proactive approaches can be difficult, or sometimes even impossible to utilize, specifically when some part of the problem remains unknown until the execution, such as external conditions, task real execution time or even tasks

in general. Therefore, reactive (online, real-time) approaches are in focus of many works ([1], [3], [6], [7], [11], [12], [14], [15], [17], [18]).

If the optimization problem is solved fast enough, a similar approach as in offline scheduling can be utilized. This can be achieved for example by approximative methods as seen in [7]. Similarly to our work, Paterna et al. in [11] created a model, which can be solved in linear time by linear programming. Even less common approach can be found in [12], where the model is solved via game theory.

Reactive approaches are commonly tied with best-effort (BE) task scheduling because BE tasks are often perceived as a source of uncertainty. We already mentioned a case when WCET and actual runtime differs and allows for reclamation. Another approach described in [14] considers the lack of strict deadline and uses Jensen's benefit function instead [9].

Change of external conditions can lead to unexpected temperature raises, for example, due to change of ambient temperature. Reactive policy to borderline temperature values can be found in [6].

Moreover, in our work, we focus on a practical evaluation of our solution. Unlike most works in the field, which are either purely theoretical or were only tested in simulations, our solution is evaluated on real hardware, in comparison to an up-to-date Linux scheduler (SCHED_DEADLINE). We found the work of Scordio et al. [17], where similar experiments were performed, as very helpful in that regard.

2.4 Previous work

Our work aims to enhance the approach from the diploma thesis of Ing. Hornof [8] and paper [4], where the main focus is Safety-critical workload. Also, DEmOS tool Section 6.1, which is utilized in both works, served us as a core part of our experiments. For the sake of continuity, we assume a similar environment and we even use the same hardware. However, the problem we are solving differs greatly. Best-effort tasks are not weighted by strict assumptions as safety-critical ones. Therefore, DVFS, preemption and task migration is fully utilized in our solution.

Chapter 3

Problem statement

To describe the problem thoroughly, we have to differentiate between the general view (see Section 3.1) and specific features concerning the actual system used for experiments (see Section 3.2). Multiple assumptions were made, regarding the model as well as tasks. Thorough description and explanation can be found in Section 3.3 The formalization of the problem follow in Section 3.4.

3.1 General problem

We assume an embedded safety-critical system responsible for periodic execution of various workloads which consist of individual tasks. Tasks are to be executed within time windows. A periodically repeating set of time windows is called a major frame.

We divide tasks into two groups, the Safety-Critical (SC) and Best-Effort (BE). SC tasks are isolated from BE tasks in their time windows, which span across all CPU cores. Due to their strict isolation, we can tackle BE task scheduling as a separate problem.

Furthermore, we assume that BE tasks are preemptive. We do not consider tasks to be dependent upon each other, nor multi-threaded. Therefore, we can approach best-effort time windows distributed across the major frame as a homogeneous time frame. We call this time frame the Best-effort Cumulative window (BEC window). The major frame and BEC window composition are illustrated in Figure 3.1.

While we assume looser conditions concerning BE tasks, which allows for postponing their execution, migration, preemption, etc., BE tasks are still assumed as a part of the system and are needed for their proper functionality. However, the execution of these tasks must not negatively influence SC tasks.

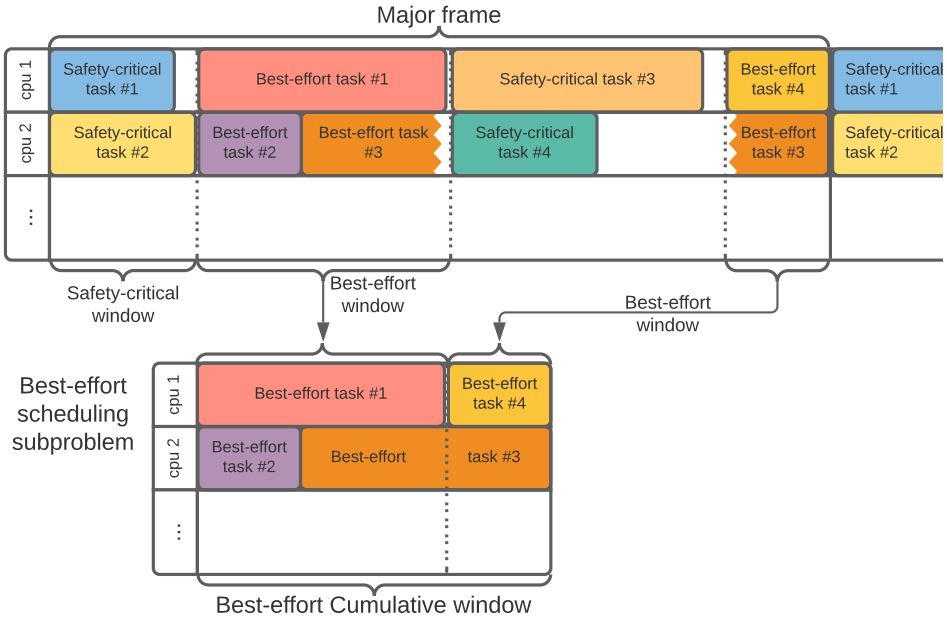


Figure 3.1: Global schedule and best-effort subproblem

Despite the software isolation, both workloads are still being executed at the same hardware and both must respect some thermal budget. Overstepping thermal boundaries (often referred to as a thermal envelope) might cause system throttling. Therefore a thermal envelope is being considered as an important condition when determining a safety-critical system.

The thermal envelope can be regarded as a shared resource. Reducing the produced heat by one part of the system allows the higher performance of the other part. Hence we target to reduce this heat via best-effort scheduling. We illustrate that dependency in Figure 3.2.

3.2 Platform

Multiple characteristics are influencing the schedule. Scheduling is dependent on the platform, upon which the schedule will be executed, therefore there are several properties we expect the platform to meet:

- The expected platform is Multiprocessor System on Chip (MPSoC).
- The expected platform may have one or more heterogeneous CPU clusters, each consisting of one or more identical CPU cores.
- The expected platform allows for Dynamic Voltage and Frequency Scaling (DVFS) per cluster.

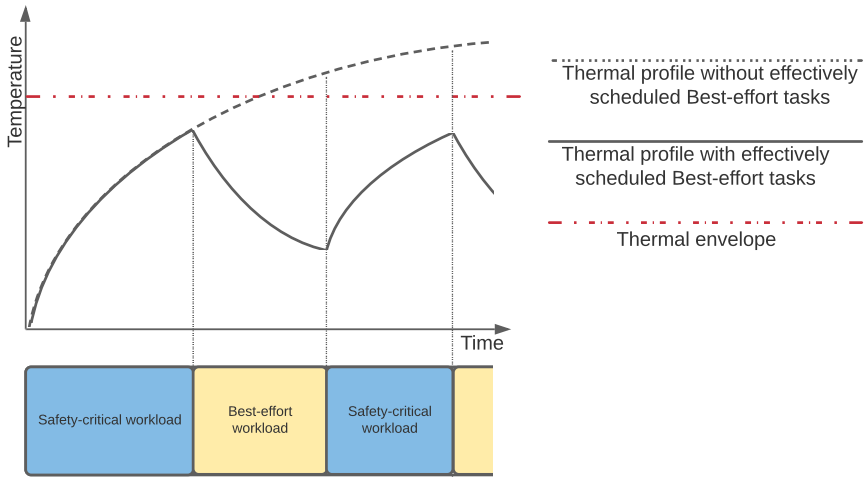


Figure 3.2: Illustration of thermal profile during task execution

- Each cluster can be set to multiple discreet frequency levels.

Note that our platform expectations are rather strict, therefore our scheduler should be usable for most of the hardware currently in use. With less strict conditions, we can adjust our scheduler, to meet them. For example, if per-core frequency scaling is available, we can approach each CPU core as a cluster consisting of only 1 core, continuous frequency scaling can be approached the high amount of discreet frequency levels, etc.

3.3 Further assumptions

To present our scheduler, multiple assumptions were made. We are deliberately assuming simple problem setting to demonstrate our ideas. We are trying to remain consistent with previous work ([4], [8]), which also leads to some assumptions to be made. Limited time and format of the diploma thesis were also a factor. Moreover, multiple assumptions were made to allow a direct comparison with the SCHED_DEADLINE scheduler. Note that some assumptions are rather strict, however can be lifted with small changes in the design of the scheduler. We discuss these options in Section 8.1.

As previously mentioned, our focus is best-effort (BE) tasks, which we consider preemptive, single-threaded and independent. These characteristics are based on the context of previous work.

Moreover, fixed affinity to the cluster for each task is assumed to allow for better comparison, because the SCHED_DEADLINE implementation does not operate well with the heterogeneous architecture. When our initial

tests were run on the hardware without specifying the cluster, the amount of work done by each task was changing erratically, which would make any comparison inconsistent. This decision also allows for solving each set of cluster assigned tasks separately, simplifying the problem; however, we are confident that our solution can solve the mixed affinity problem as well (see Section 8.1).

The last task-related assumption influences start time and deadline. We require each task to be executed during the BEC window, but we do not make further demands on specific deadlines or star times of each task. BE task scheduling is often tied up with soft deadlines and therefore we do not focus on removing this particular constrain. Moreover, we would have to respect the location of BEC window partitions within the major window when estimating start times and deadlines. However, we think that deadlines and start times could be added by further dividing the BEC window into smaller parts (see Section 8.1).

3.4 Formalization

We model previously mentioned problem, platform and assumptions. First of all we need to describe physical capabilities of used MPSoC:

- We assume a set of m CPU clusters $\mathcal{C} = \{c_1, ..., c_m\}$.
- For each cluster c_j, there is a finite ordered set containing n possible frequency settings $\mathcal{F}_j = \{f_1, ..., f_n\}$, $f_1 < f_2 < ... < f_n$.
- Each cluster c_j consists of K_j identical cores.
- Each cluster c_j has set $\mathcal{U}_j$ of possible core utilizations (number of active cores), $\mathcal{U}_j = \{1, ..., K_j\}$.
- We define "CPU settings" $\mathcal{S}_j$ as a set of unique combinations of core utilization and frequency setting. Note, that there is a special case of CPU idle state s_{idle}. $\mathcal{S}_j = \{\mathcal{U}_j \times \mathcal{F}_j\} \cup \{s_{idle}\}$.
- We use f_s as a reference to frequency of given $s \in \mathcal{S}_j$.
- We use u_s as a reference to core utilization of given $s \in \mathcal{S}_j$.

The problem is divided between clusters and each cluster is solved separately. Results of each separate problem are later merged into the final schedule. We describe our tasks in the following manner:

- We assume a set of BE tasks $\mathcal{T}_j$ for each cluster c_j. Each task denoted as $\tau \in \mathcal{T}_j$.

- Each task $\tau \in \mathcal{T}_j$ has a prescribed amount of work W_τ.
- For each task $\tau \in \mathcal{T}_j$ we measured efficiency coefficient $E_{\tau,f}$, which describes how well it performs under frequency f in comparison to f_n.
- For each task $\tau \in \mathcal{T}_j$ we measured average power consumption of the platform under the given task execution $\pi_{\tau,s}$ on each CPU setting $s \in \mathcal{S}_j$.

We need to formalize external conditions:

- We denote the length of BEC window by L.
- We denote the value of temperature threshold (thermal envelope) as Δ.

The outputs (variables) of our model are time constraints describing each window and task allocation:

- We devide BEC window into intervals of length a_s for each CPU setting $s \in \mathcal{S}_j$.
- The runtime of each task $\tau \in \mathcal{T}_j$ is then divided into these windows, where it will run for time interval $d_{\tau,s}$ for each $s \in \mathcal{S}_j \setminus s_{idle}$. Note that in the idle state s_{idle}, no task is being executed.

Chapter 4

Scheduler

To solve the problem of BE task scheduling under thermal constraints, we propose the offline scheduler. Our scheduler works in two phases. In the first phase, the problem is decomposed per cluster, modelled as an optimization problem and solved separately. In the second phase, we parse results from models. We create per-cluster schedules via McNaughton's algorithm and then we merge separated schedules into the final schedule.

The most important part of each optimization problem is the objective function, which represents the goal of the optimization process. Our main goal is to lower the temperature of the whole system. However, temperature modelling is very complex, while we strive for an approximative description, which can be easily solved even for big instances. Therefore, we use to our advantage measured relation between the power and relative temperature, which relies on some assumptions (e.g. the length of hyper period), from [4]. In Figure 4.1 we can clearly observe the linear relation between the average power consumption and the steady-state temperature [4].

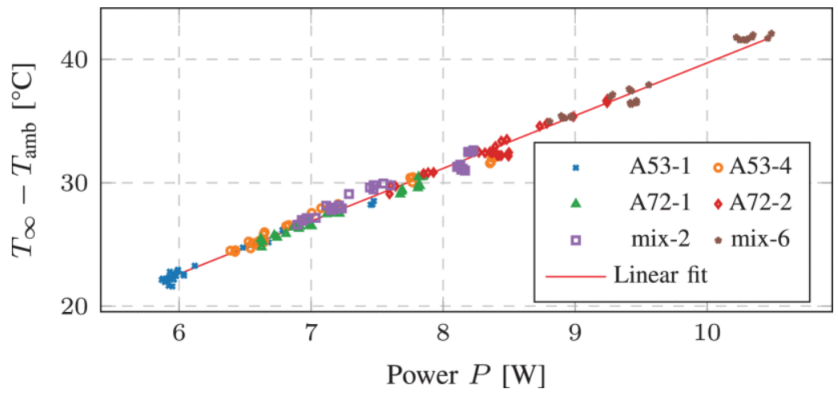


Figure 4.1: Relation between average power consumption P and relative temperature (i.e., difference between steady state temperature T_inf and ambient temperature T_amb)

4.1 Model

When any workload execution is inspected, we can describe it as a series of time intervals, where each time interval has some frequency and core utilization setting. We call this combination of frequency and CPU core utilization a "CPU setting". Switching between "CPU settings" and especially between frequencies creates overhead, which we strive to lower, to make the schedule more effective. Therefore, we aggregate those small intervals into larger time windows of length a_s. Moreover, we order these time windows by frequency to lower frequency switching overhead even further. We illustrate our schedule structure in Figure 4.2, where a_1 marks the first window of frequency f_1 and core utilization $u_1 = 1$.

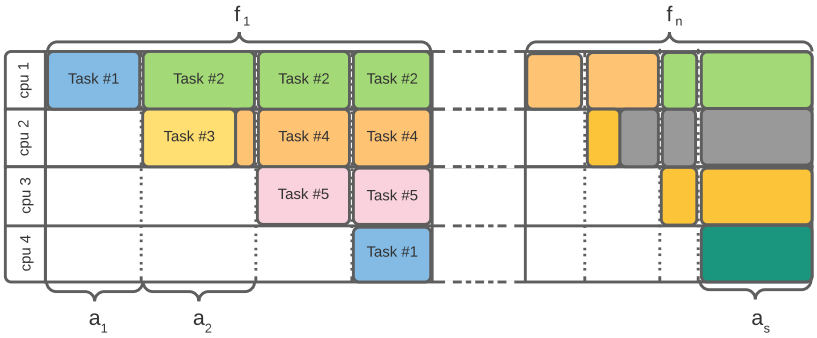


Figure 4.2: Illustration of settings

Both frequency and core utilization have an impact on power consumption, therefore for each CPU setting $s \in \mathcal{S}_j$, we can measure some average power consumption P_s. Because of the linear relation, we can use average power consumption in the objective function instead of temperature. Minimizing the overall power consumption will lead to minimizing the temperature as well.

$$\min \sum_{s \in \mathcal{S}_j} P_s \cdot a_s \tag{4.1}$$

We observed (see Section 5.2.2) that the average overall task is not very precise. Some tasks have much higher power consumption $\pi_{\tau,s}$ than others. Therefore we wanted to use a weighted average ϵ_s for each window.

$$\min \sum_{s \in \mathcal{S}_j \setminus s_{idle}} \epsilon_s \cdot a_s + P_{s_{idle}} \cdot a_{s_{idle}} \tag{4.2}$$

$$\text{where: } \epsilon_s = \sum_{\tau \in \mathcal{T}} \frac{\pi_{\tau,s} \cdot d_{\tau,s}}{u_s \cdot a_s} \tag{4.3}$$

We can substitute:

$$\lambda_{\tau,s} = \frac{\pi_{\tau,s}}{u_s} \tag{4.4}$$

After substitution Equation (4.4) into the equation 4.2, we can simplify the equation into following objective function:

$$\min \sum_{s \in \mathcal{S}_j \setminus s_{idle}} \sum_{\tau \in \mathcal{T}} \lambda_{\tau,s} \cdot d_{\tau,s} + P_{s_{idle}} \cdot a_{s_{idle}} \tag{4.5}$$

Note that $\lambda_{\tau,s}$ and $P_{s_{idle}}$ are constants, therefore the objective function Equation (4.5) is linear.

The whole BEC window of length L is divided into windows, therefore:

$$\sum_{s \in \mathcal{S}_j} a_s = L \tag{4.6}$$

No task can run in the window s longer than is the length of the window itself, therefore:

$$d_{\tau,s} \leq a_s \quad \forall \tau \in \mathcal{T}, \forall s \in \mathcal{S}_j \tag{4.7}$$

Also in each window can only be executed as much work as the length of the window and amount of core running allows:

$$\sum_{\tau \in \mathcal{T}} d_{\tau,s} \leq u_s \cdot a_s \quad \forall s \in \mathcal{S}_j \tag{4.8}$$

In Section 3.4 we denoted W_τ as an amount of prescribed work. We quantify W_τ as an equivalent of the runtime of the task τ under the highest possible frequency. Runtime intervals $d_{\tau,s}$ are scaled by efficiency ratio E_{τ,f_s}, to obtain the maximum frequency runtime equivalent.

$$\sum_{s \in \mathcal{S}_j} d_{\tau,s} \cdot E_{\tau,f_s} = W_\tau \quad \forall \tau \in \mathcal{T} \tag{4.9}$$

All constants and variables are assumed not to be negative. While this might seem like a strong assumption from a mathematical point of view, it is only natural, that we do not assume negative time, power or effectiveness. Our proposed model as amended is linear with continuous variables and thus it can be solved in linear time by linear programming (LP) solver.

Note that our model is **aproximative**. The average power consumption presented is an upper bound, which presents the case of 100% CPU bandwidth being utilized. Measured constants present averages over limited time periods and fixed temperatures. There are additional time and power demands on task migration and frequency switching which we deliberately omit in our model, due to complexity. Our experiments are conducted on a device with an operating system and we can not prevent system operations during the scheduling.

4.2 Creating the schedule

The result of the model is not only the value of the objective function, which in our case is estimated average power consumption but values of our variables a_s and $d_{\tau,s}$ which describe the length of each window corresponding to each CPU setting $s \in \mathcal{S}_j$ and length of execution interval of each task in given window respectively.

In order to create the schedule, we have to prepare time windows of length a_s, with a given "CPU setting" s, which we order by frequency. For each window, we allocate execution intervals for each task, given by $d_{\tau,s}$. Because our tasks are preemptive and can be migrated between cores, to allocate intervals, we utilize McNaughton's algorithm (which we illustrate in Figure 4.3).

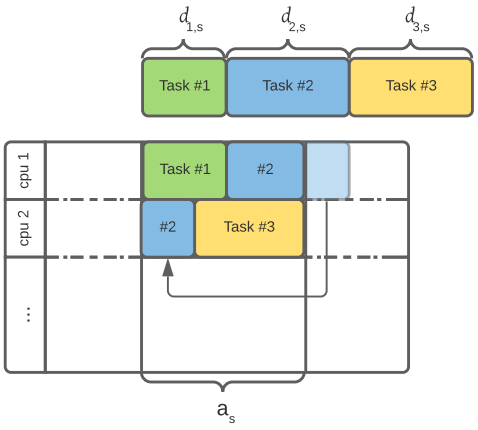


Figure 4.3: Task to core allocation

Once time windows are created for each cluster, we merge solutions together. In order to reduce temperature spikes, we order windows of each cluster in a different manner. Windows with high frequency will produce more heat than

those with lower frequency, thus we want to prevent the simultaneous run of the highest frequencies on all clusters.

4.3 Reactive policies

Our scheduler works as long as the problem is solvable and it tries to lower the average power consumption and thus lowers the temperature. However, in real-world applications rather than a demand for lowering the temperature, a strict thermal envelope Δ is given. With a given amount of work, a feasible solution might not exist. Moreover, external conditions (e.g. change of ambient temperature) might influence the validity of the solution and the scheduler in a critical system must be able to reflect it. Therefore, we propose three reactive policies.

Thermal throttling in case of overstepping thermal boundaries is present in most of today systems. Commonly, in the case of temperatures over the threshold on a CPU, a CPU frequency is lowered. This behaviour is reflected by our first DVFS policy, which we call "soft throttle".

Our second policy is called the "skip" policy and its control of temperatures is based on imminent lowering of executed workload. Tasks are skipped in case of overheating.

The final policy also strives to achieve temperature control through workload reduction, however in a more elaborate way. Instead of skipping individual tasks, we try to utilize a new reduced schedule.

There are three aspects we focus on. Besides the temperature control, we also want our solution to achieve the best possible efficiency. Therefore instead of flat reduction of frequency or workload, we try to lower them gradually. Our goal is to not only create flexible policies with high efficiency, but we also want our policies to be "fair" in the sense that the ratio between tasks is preserved.

Part II

Implementation and experiments

Chapter 5

Environment

We hold real experiments in high regard. Therefore, we focus on their reproducibility. We describe the environment and tools in great detail. We also report all our experiments' settings and raw measured data besides processed data.

We describe our hardware used for all experiments in Section 5.1. The device runs the operating system Debian GNU/Linux 10 (Buster). We can not prevent system operations during our measurements, for that reason, all our measurements are weighted by error. We run all our experiments in multiple cycles or in multiple separated runs and for long time periods to diminish the influence of unexpected OS operations and other external conditions.

Our model depends on multiple hardware-dependent constants, which we summarize in Section 5.2. On the software side, we used Linux operating system. Its scheduler served us as a baseline for our experiments. More detailed description is in Section 5.3.

5.1 Hardware

Our platform used for experiments is i.MX 8QuadMax Multisensory Enablement Kit (MEK).

The i.MX8 board implements ARM's big.LITTLE architecture. Big.LITTLE technology is a heterogeneous processing architecture that uses two types of processors. "LITTLE" processors are designed for maximum power efficiency while "big" processors are designed to provide maximum compute performance. [21]

The "LITTLE" ARM Cortex-A53 CPU cluster consists of 4 cores, offering frequencies 600 MHz, 896 MHz, 1104 MHz and 1200 MHz. The "big" ARM

Cortex-A72 CPU cluster consists of 2 cores, offering frequencies 600 MHz, 1056 MHz, 1296 MHz and 1596 MHz. The frequency of each cluster is always the same for all of its cores.

The temperature measurements were provided by built-in sensors of i.MX8 chip. We also used HTU21D ambient temperature sensor, which is not part of the i.MX8, but we consider it as a vital part of our test-bed, as well as the power consumption measurement module INA219 [28].

We specify formalization from Section 3.4 as follows.

- $\mathcal{C} = \{a53, a72\}$
- $\mathcal{F}_{a53} = \{600 \text{ MHz}, 896 \text{ MHz}, 1104 \text{ MHz}, 1200 \text{ MHz}\}$
- $\mathcal{F}_{a72} = \{600 \text{ MHz}, 1056 \text{ MHz}, 1296 \text{ MHz}, 1596 \text{ MHz}\}$
- $\mathcal{U}_{a53} = \{1, 2, 3, 4\}$
- $\mathcal{U}_{a72} = \{1, 2\}$

The specification can be also found in processor.json file, which contains the processor settings for our scheduler.

5.2 Hardware dependent constants

Constants $E_{\tau,f}$ and $\pi_{\tau,s}$ used in Section 3.4 has to be measured to reflect the nature of the hardware and tasks. We use benchmarks to represent our tasks.

5.2.1 Efficiency coefficient

The first parameter which needs to be measured is task efficiency $E_{\tau,f}$. We use the highest available frequency as a baseline (1200 MHz for A53, 1596 MHz for A72). Thermobench [23](described in detail in Section 6.2) is used to collect data from each benchmark. We estimated iterations done per second and calculated the ratio $E_{\tau,f}$, for each task, for each cluster, as seen in Table 5.1 and Table 5.2 respectively. Detailed results can be found in the attached file: task_efficiency_measurements.ods. Measured values are also part of the benchmark specification file benchmarks.json.

Even though we did not observe a big fluctuation in efficiency scaling in the selected group of benchmarks, we consider using separate values as a good practice (instead of using an average for example), because some specific tasks might not scale as linearly as shown. For example tasks with

Cluster:	A53			
Frequency:	600 MHz	896 MHz	1104 MHz	1200 MHz
a2time	50.69 %	75.38 %	92.46 %	100 %
aifirf	50.02 %	74.67 %	91.97 %	100 %
bitmnp	49.93 %	74.68 %	92.01 %	100 %
candrdr	50.36 %	74.98 %	92.19 %	100 %
idctrn	49.92 %	74.60 %	91.95 %	100 %
iirflt	50.00 %	74.69 %	92.13 %	100 %
matrix	50.98 %	75.68 %	92.60 %	100 %
pntrch	49.97 %	74.69 %	92.02 %	100 %
puwmod	49.95 %	74.67 %	92.06 %	100 %
rspeed	51.87 %	76.25 %	92.45 %	100 %
tblook	50.28 %	75.20 %	92.44 %	100 %
ttsprk	49.89 %	74.61 %	91.97 %	100 %
membench	90.04 %	95.12 %	98.37 %	100 %

Table 5.1: Efficiency measurements A53

Cluster:	A72			
Frequency:	600 MHz	1056 MHz	1296 MHz	1596 MHz
a2time	37.53 %	66.08 %	81.22 %	100 %
aifirf	37.47 %	66.04 %	81.19 %	100 %
bitmnp	37.38 %	66.03 %	81.23 %	100 %
candrdr	37.57 %	66.11 %	81.35 %	100 %
idctrn	37.45 %	65.99 %	81.17 %	100 %
iirflt	37.45 %	65.95 %	81.21 %	100 %
matrix	40.16 %	68.17 %	82.61 %	100 %
pntrch	37.40 %	65.88 %	80.95 %	100 %
puwmod	37.46 %	65.96 %	81.20 %	100 %
rspeed	37.49 %	65.96 %	81.21 %	100 %
tblook	37.40 %	65.73 %	81.08 %	100 %
ttsprk	37.59 %	66.01 %	81.12 %	100 %
membench	79.86 %	92.21 %	96.04 %	100 %

Table 5.2: Efficiency measurements A72

high memory usage will not scale well with higher CPU frequency due to the bottleneck in memory accesses. We present this behaviour by synthetic benchmark membench, which has a high amount of memory access. Poor scaling is clearly visible from measured results in last rows of Table 5.1 and Table 5.2.

Moreover, using specific $E_{\tau,f}$ for each task does not increase computational complexity because the $E_{\tau,f}$ values figure in our model as constants.

5.2.2 Power consumption

The second hardware dependent constant is average power consumption $\pi_{\tau,s}$. For each CPU setting s on each cluster (A53, A72) we conducted a series of experiments, measuring benchmarks' (tasks') average power consumption for 100 seconds. We sample the power consumption with period of 1 second, the provided samples are averages over that period. Since we use 2 clusters, with 8 and 16 CPU settings respectively and 12 different benchmarks, we had to conduct 288 measurements ($(8 + 16) \times 12$). Detailed measurements can be found in the attached file: task_efficiency_measurements.ods and summarized results in benchmarks.json.

Due to observation in Equation (4.4), we use $\lambda_{\tau,s}$. We show an example of computed values of $\lambda_{a2time,s}$ in Table 5.3.

A53				
Cores:	1	2	3	4
600 MHz	6.05 W	3.11 W	2.17 W	1.62 W
896 MHz	6.13 W	3.17 W	2.23 W	1.69 W
1104 MHz	6.41 W	3.35 W	2.32 W	1.81 W
1200 MHz	6.55 W	3.44 W	2.40 W	1.87 W
A72				
Cores:	1	2		
600 MHz	6.48 W	3.41 W		
1056 MHz	6.78 W	3.68 W		
1296 MHz	7.10 W	3.92 W		
1596 MHz	7.61 W	4.37 W		

Table 5.3: Average power consumption measurements per core for a2time benchmark

Measurements are in line with our expectation of energy consumption growing with higher frequency and with higher core utilization. Note that $\lambda_{\tau,s}$ values presented are $\pi_{\tau,s}$ divided by the utilized core count.

In our model, we use $\pi_{\tau,s}$ values in weighted average (see Section 4.1). We are aware that a combination of different tasks running at the same time on the chip may result in values varying from the weighted average, however testing all combinations of tasks would lead to $|\mathcal{U}| \times \sum_{i=0}^{K} |\mathcal{T}|^i$ combinations, which in our case is over 90 000 combinations for A53 cluster alone, where each combination would need its measurement for at least several benchmark iterations to estimate power consumption with reasonable precision. If we assume the same period of 100 seconds, it would take over 100 days to conduct such an experiment. Such a level of detail would make the solution unpractical for use while we do not expect any major improvement of the solution. Moreover, a different mathematical model would have to be used, which would need logical variables and thus it would lose its linear properties

(it would lead to a mixed-integer programming which has higher complexity).

5.3 Linux scheduler SCHED_DEADLINE

CPU scheduler is a vital part of every operating system. Linux systems nowadays offer multiple different schedulers, such as SCHED_FIFO, SCHED_OTHER, SCHED_RR, etc. We chose SCHED_DEADLINE for comparison due to multiple reasons. Mainly it reflects our use case, where tasks are periodically executed with firm deadlines, runtimes and periods.

The SCHED_DEADLINE scheduler is built upon two algorithms. Earliest Deadline First (EDF) algorithm runs the task with the closest deadline first. This algorithm can schedule tasks up to 100% utilization of a single processor; however, its guarantees drop significantly for a multi-processor problem. Therefore for multi-processor usage, a variant of global EDF is used, which has better guarantees, especially in a case where period and deadline is identical (which is our case) [29].

The second part of the scheduler is based on Constant Bandwidth Server (CBS), which provides protection for real-time sporadic tasks during the scheduling. Moreover, SCHEAD_DEADLINE can be enhanced by the Greedy Reclamation of Unused Bandwidth (GRUB) algorithm, which allows for CPU reclamation of unused bandwidth; however, bandwidth reclamation is needed, when a task finishes sooner than its Worst-Case Execution Time (WCET) estimates. While this is a rather common event in a real system, we do not assume this to happen. Our tasks are run in infinite loops, utilizing all system time given by the scheduler. The algorithm does not allow for WCET overstepping and in such case, the task is throttled until the next period.

We conducted our baseline experiments via Linux tool chrt [25]. Chrt is a ready-to-use command-line tool that allows for easy scheduling policy assignment to a process. Unfortunately, its lightweight design does not allow for setting system flags, which among other features decide which reclamation algorithm is used (CBS is the default choice).

The SCHED_DEADLINE also has multiple strict limitations, which we had to respect when creating our test cases and presenting measured values. The SCHED_DEADLINE does not allow processes to fork. All processes scheduled by it have the same priority level 0. On one hand, this allows our experiments to run with less interruption (only system-level tasks has higher priority), but on the other side, it also poses a disadvantage in means of measuring. Multiple metrics such as the CPU temperature and frequency are being gathered via the Thermbench tool (see Section 6.2), which is getting limited in the case of high CPU usage.

Chapter 6

Implementation and tools

Multiple tools were used during the preparation and execution of our experiments. In spite of the reproducibility of our work, we reference all external tools and we include all scripts used in the attached files.

We describe DEmOS, the vital tool for scheduled task execution in Section 6.1. We also want to highlight Thermobench, the main tool for all measurements in Section 6.2. We refer to the benchmark suite Autobench substituting our tasks in Section 6.3.

Finally, we talk about our own work, the scheduler and scheduling policies in Section 6.4 and Section 6.5 respectively.

6.1 DEmOS

DEmOS is an open-source tool for scheduling in the Linux environment [22]. It takes care of resource allocation for tasks specified in the YAML-formated configuration file. The configuration is divided into two parts. The first part contains the schedule. The schedule is described as a periodically repeating sequence of windows. Each window has a time budget and consists of slices for each CPU cluster available. Both SC and BE partitions can be present in slices; however, we only use BE partitions in our case. BE partitions also may contain frequency settings. In that case, a special "imx8_per_slice" policy (or any of our reactive policies) must be defined to take the frequency setting into account.

Partitions encapsulate one or more processes and they form the second part of the configuration file. Each process describes the task to be executed. Processes are stored and run as shell commands. The time budget is also part of the process description.

DEmOS has been used for our experiments with both online and offline

scheduling. Our scheduler outputs DEmOS YAML configurations. Reactive policies we propose are implemented as a part of the DEmOS tool.

6.2 Thermobench

Thermobench [23] is open-source software for capturing execution profiles of user-defined workloads [19]. It allows for gathering multiple statistics during the run of a task and it was a vital tool for all our experiments. Thermobench allows for gathering not only temperatures on cores but also program outputs, CPU load, CPU frequencies and characteristics gathered by external devices, such as ambient temperature and power consumption. Moreover, all data are automatically timestamped.

Unfortunately, the thermobench tool creates some overhead. While this overhead does not negatively influence our comparison experiment, because it is run in both cases (for our scheduler as well as for SCHED_DEADLINE), there is a lack of consistency at SCHED_DEADLINE experiments, in cases where utilization of CPU bandwidth is very high.

6.3 Autobench 2.0

The vital part of most experiments in programming is a benchmark suite. We use benchmarks to simulate our best-effort tasks. Tasks must be preemptive and allow for easy observability. After initial attempts with MiBench [20], which were very helpful to get familiar with the environment due to their implementation simplicity, we decided to opt for AutoBench 2.0. [24]. While both suites follow the Embedded Microprocessor Benchmark Consortium (EEMBC) model, previous works at thermobench [19] allow usage of a wrapper for easier control of Autobench iterations.

We used 12 benchmarks to represent our best-effort tasks (see e.g. Table 5.1). Each benchmark is being executed in an endless loop. Thanks to the thermobench wrapper, we can easily track iterations done for the final comparison of completed work. However, one iteration of each benchmark takes a different amount of time. To make a comparison between benchmarks simpler, we measured the average time of one iteration of each benchmark and then we set control prints accordingly, so every benchmark would report after the same amount of execution time.

6.4 Scheduler

We create a command-line tool for problem generating and scheduling. Our implementation is done in Python programming language and can be found in attached files, in scheduler directory. The user entry-point is generator.py, to list of all possible settings, run the program with option `--help`. We illustrate the structure of our program in Figure 6.1.

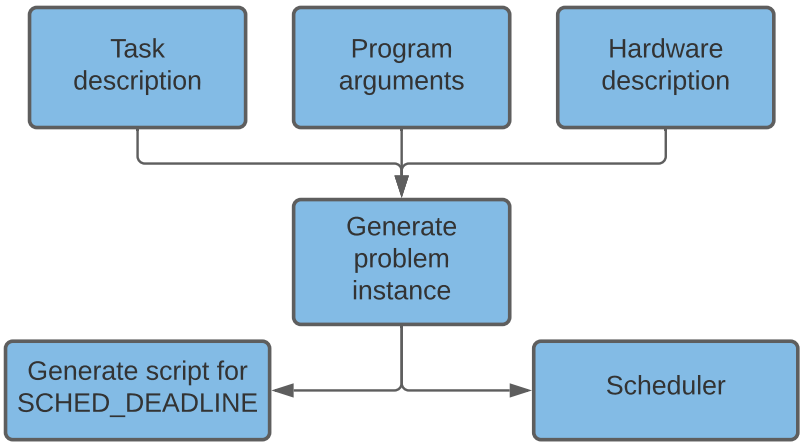


Figure 6.1: Scheme of our instance generating and solving program

There are three main input points for our program. The first is the hardware description file (for detailed structure see processor.json in attached files). Our program does not rely on specific hardware and in theory, it could be usable for other CPU architectures, but note that the final schedule is created as a YAML configuration file for DEmOS, where portability might be an issue. The hardware description file contains basic information about the target platform with a separate description of each cluster, including numbering of cores, available frequencies and name. The name of the cluster can be chosen arbitrary, but for usage with SCHED_DEADLINE (due to lack of affinity setting described in Section 5.3) we have to create CPU groups in advance. Names of these groups must be the same as the names we chose for clusters.

The second input is the task description file (for detailed structure see benchmarks.json in attached files). The benchmark file must contain a list of all benchmarks with their hardware-dependent constants (see Section 5.2). Note that the structure must respect the hardware description file. Furthermore, running benchmarks (tasks) might be again platform-dependent (e.g. on a different platform, a different file structure can be expected, leading to a different command being used for running the given benchmark). Therefore, we created a *Task* abstract class in task.py, which needs to be implemented for usage on a different platform (for example see our *Autobench* class in the

same file).

Finally, command line input options are gathered. These options influence problem instance generation.

6.4.1 Generating problem instance

Problem instances are generated separately for each cluster, since each cluster is solved separately. We can formalize the problem instance for cluster j as $mathcal I_j = \langle L, \mathcal{T}_j, \mathcal{W}_j \rangle$, where L is the length of the BEC window, $\mathcal{T}_j$ is set of tasks τ for given cluster and $\mathcal{W}_j$ contains workloads of tasks allocated to cluster j ($\mathcal{W}_j = \{W_\tau | \forall \tau \in \mathcal{T}_j\}$).

While it would be possible to create problem instances manually, we do not consider this option in our program because we want to test our scheduler with intricate problems considering tens of tasks, which would be tedious to create by hand. Therefore, we create an automatic problem generator, which creates problem instances based on multiple values:

- Length of the BEC window L (`--period`) set in milliseconds.
- Core count of the cluster K (loaded from hardware description file).
- CPU usage U (`--usage`). CPU usage serves as an estimate of CPU load. $\sum_{\tau \in \mathcal{T}} W_\tau = L \cdot K \cdot U$
- Maximum task runtime M (`--maximum`). $W_\tau \leq M : \forall \tau \in \mathcal{T}_j$
- Minimum task runtime N (`--minimum`). $W_\tau \geq N : \forall \tau \in \{\mathcal{T}_j \setminus \tau_{last}\}$. The minimum value might be violated at last task τ_{last} to fit bandwidth given by L, K and U.
- Seed S (`--seed`). Tasks and their runtimes are chosen pseudo-randomly based on seed value, to allow for replication of the same instance.

Our algorithm first quantifies the CPU usage in milliseconds as a bandwidth B ($B = \sum_{\tau \in \mathcal{T}} W_\tau = L \cdot K \cdot U$). Tasks are then chosen pseudo-randomly from $\mathcal{T}$ ($\mathcal{T}$ is determined by task description file), with pseudo-randomly given runtimes to fill the given bandwidth. Due to the randomness of selecting tasks and their workloads, the number of tasks might vary for the same instance setting with a different seed. We demonstrate our approach by Algorithm 1.

With work assigned to each randomly chosen task, for each cluster, the problem instance is complete. The instance then serves as an input for the SCHED_DEADLINE script generator (`bash.py`), which creates bash scripts. This bash script simply runs all tasks with a given setting under SCHED_DEADLINE by Linux chrt tool. Finally, the problem instance is also the main input of our scheduler.

Algorithm 1 Generating the problem instance for cluster j

$B \leftarrow K \cdot L \cdot U$
$\mathcal{T}_j \leftarrow \{\}$
$\mathcal{W}_j \leftarrow \{\}$
while $B > 0$ **do**
  $\tau \leftarrow$ pseudo-random choice from $\mathcal{T}$
  if $B > M$ **then**
    $W_\tau \leftarrow$ pseudo-random choice from interval $[N, M]$
  else
    $W_\tau \leftarrow B$
  end if
  $B \leftarrow B - W_\tau$
  $\mathcal{T}_j \leftarrow \{\mathcal{T}_j \cup \tau\}$
  $\mathcal{W}_j \leftarrow \{\mathcal{W}_j \cup W_\tau\}$
end while
return $\langle L, \mathcal{T}_j, \mathcal{W}_j \rangle$

6.4.2 Scheduling

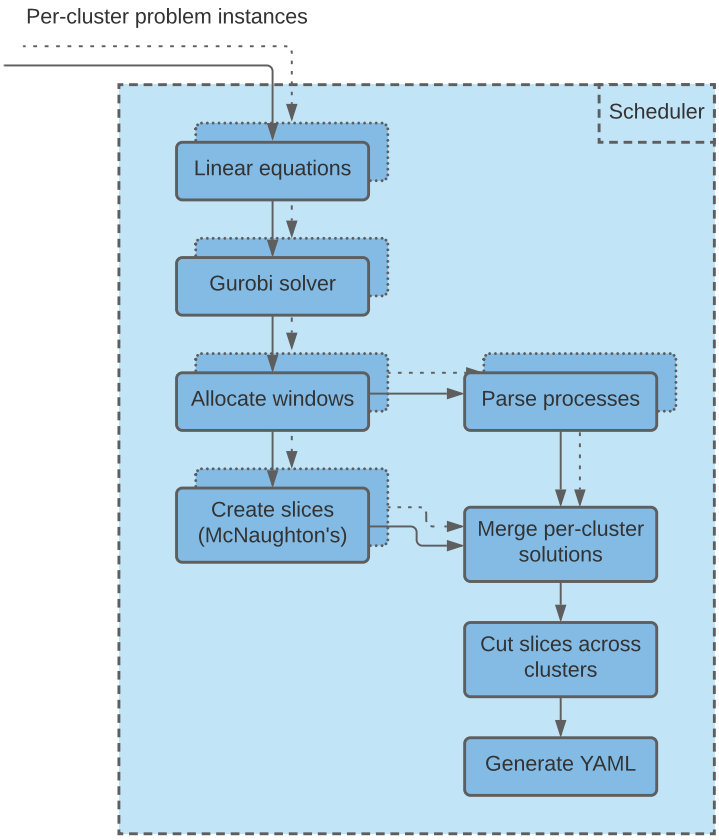


Figure 6.2: Scheme of our scheduler

We illustrate the run of the scheduler in Figure 6.2. In this case, we assume multiple clusters, where each problem is solved separately and the solution is merged in the end. In the case of a single CPU cluster, YAML configuration is created after application of McNaughton's algorithm (there is no need for merging and cutting across clusters).

The scheduling process starts separately for each cluster. In the first phase, the scheduler creates a set of linear equations (constraints and the objective function as seen in Section 3.4). Then Gurobi optimizer is called to solve the resulting Linear Program (LP). The Gurobi Optimizer is a mathematical programming solver available for multiple languages, including libraries in Python.

After solving the set of linear equations, we finally obtain values of a_s and $d_{\tau,s}$. From $d_{\tau,s}$ we can create processes for the final YAML configuration file (we need a sum of $d_{\tau,s}$ for setting the budget of each task). Variables a_s allow us to allocate the windows. If for some window defined by $s \in \mathcal{S}_j : a_s = 0$, we can skip this window completely. For all windows of non-zero length, we call McNaughton's algorithm to fill these windows with task intervals $d_{\tau,s}$. We illustrate this process in Figure 6.3.

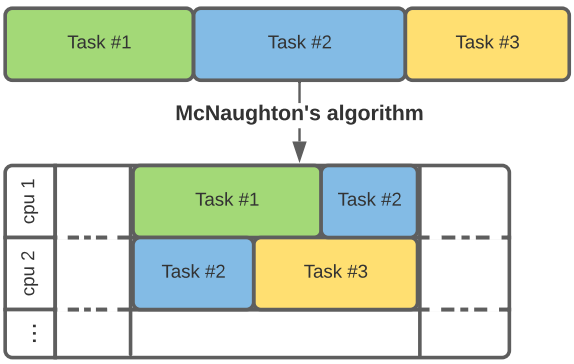


Figure 6.3: Scheme of McNaughton's algorithm

Moreover, we cut windows by delimiters obtained for each core, to create slices as also shown in Figure 6.4. We need slices to represent the YAML configuration settings for DEmOS. Each slice contains a unique combination of tasks and frequency.

Once processes and sliced windows for each cluster are prepared, we can finally merge our solution into the final schedule. The merging of processes is straightforward (by simply appending); however, merging windows is a bit more intricate. First of all, the frequency setting for each cluster is independent of each other. Therefore, we switch the order of frequencies at the second cluster, from ascending to descending order of frequency. We reduce the thermal spikes by pairing a low frequency of one cluster with a high frequency of the other one. Then we finally merge windows of both

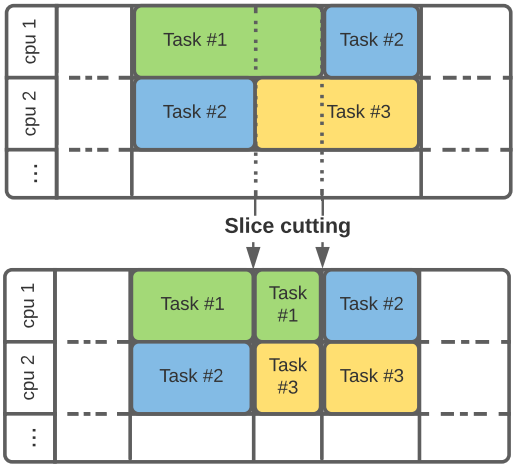


Figure 6.4: Scheme of creating slices

clusters. New slices are made again, as previously because some delimiters between slices in each cluster are likely to be different. We illustrate a simple case in Figure 6.5. The schedule is exported to the YAML configuration afterwards.

One very promising feature of our mathematical model is linearity. In our definition of the problem, all variables are continuous and all of our equations are linear. Therefore, the problem rates as an LP problem. LP can be solved in polynomial time. This means, that our scheduler is potent to solve even large instances in a reasonable time.

We enlarged potential problem instances by an increasing amount of tasks and we measured the complete runtime of the program. In Table 6.1 we can observe, that our scheduler is capable of solving instances of hundreds of tasks in less than a second. Overall we estimated polynomial complexity growth $\mathcal{O}(n^2)$ of our scheduler. Solving the LP alone was scaling even better, but despite better results, the complexity of the Gurobi LP solver is still estimated as a polynomial.

tasks	8	54	114	555	1107	2776	5541
runtime [ms]	34.34	62.74	108.95	512.25	1 510	5 340	16 590
Gurobi [ms]	2.17	5.14	11.02	100.58	422.29	1 290	2 670

Table 6.1: Scheduler runtime measurements

While the overhead of the scheduler can be considered too big for real-time usage, we focus on offline scheduling and therefore our implementation is imperfect. More overhead is caused by file creation for experiments, preparation of the problem instance and some steps in our program are needed for the DEmOS interface, but not for scheduling itself. However, our mathematical model allows for a fast solution.

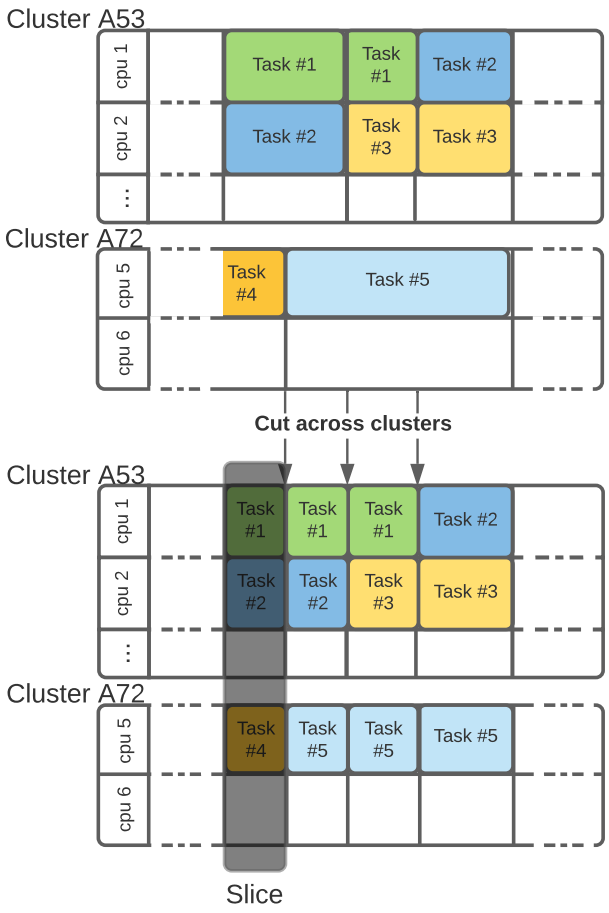


Figure 6.5: Scheme of cutting slices across cluster schedules during merging

6.5 Reactive policies

We use the schedule precomputed by our scheduler for the main run. On top of that, our reactive policies gather temperature readings from the CPU and measure an average temperature over the recent period. As previously mentioned, we want our policies to act gradually, to achieve high efficiency while respecting the thermal envelope. Therefore, we introduce policy levels.

Based on the temperature average, the policy level is set accordingly. Higher policy level forces more strict rules. Each policy reacts to the policy level in a different way. We depict the workflow in Figure 6.6. In the case of policy level 0, all policies execute the schedule without any changes.

While policy level change influences schedule execution instantly, the temperature will change with a delay due to the thermal capacity of the chip. To prevent overreacting to a temperature growth, we create a cooldown timer,

which blocks further policy-level changes for a selected period of time.

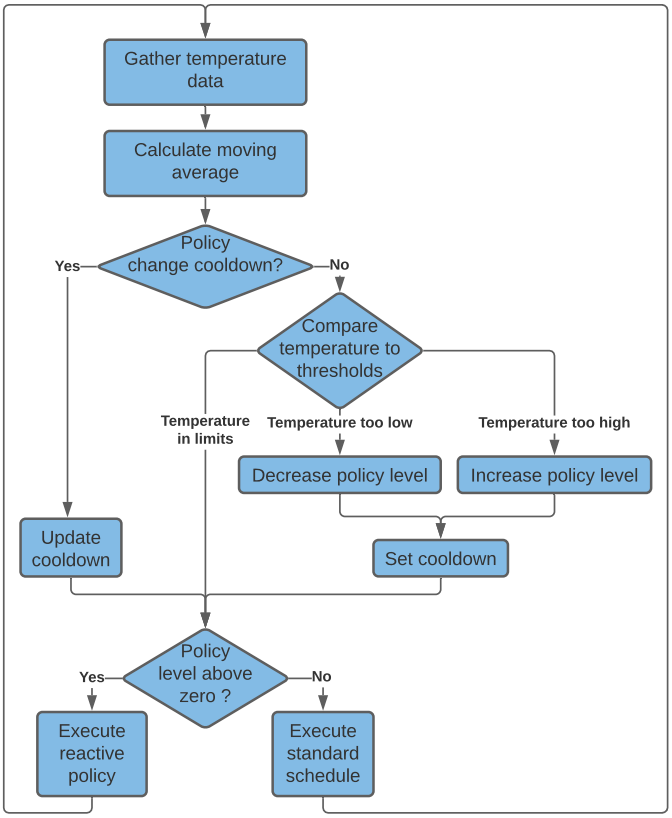


Figure 6.6: Scheme of reactive policy

Our first policy is called the "soft throttle" policy and it utilizes DVFS. Unlike strict frequency throttling often deployed in today CPUs, which lowers frequency to the minimum level, our policy reduces the frequency gradually, until thermal boundaries are reached. We lower all tasks frequencies level by level ($f_n \rightarrow f_{n-1}$), in order to preserve the bandwidth ratio of each window, thus preserving fairness. In our case, our hardware offers 4 frequencies, therefore there are up to 4 levels of reaction.

The second reactive policy takes advantage of core utilization by task skipping. In case of overheating, task execution is skipped, leaving thus some core in an idle state, which is thermally less demanding. In order to sustain the fairness of the solution, the task to be skipped are chosen in a round-robin manner. The number of skipped tasks at any time grows with policy level (e.g. policy level 2 means that 2 tasks are skipped in each slice, thus leaving 2 cores idle).

The last proposed policy utilizes offline scheduling. In advance we create multiple schedules, gradually scaling down the total work amount W_τ for each task $\tau \in \mathcal{T}_j$. This way we achieve an ideal task execution ratio. However,

switching between multiple schedules is not possible at the moment due to the implementation of DEmOS, which is used to execute all our schedules and policies. Therefore, we use a static level for our comparison tests, leaving the actual reactive aspect of the proposed policy for future work.

Chapter 7

Experiments

In this chapter, we describe our experiments and results. Experiments on real hardware are rather rare in the field; however, we believe that practical experiments are the best way to prove our approach. Moreover, we pay close attention to the experimental setting, since we strive to make our experiments reproducible. We tested proactive and reactive approaches separately.

7.1 Proactive scheduling

7.1.1 Experimental setup

We implemented our scheduler as a part of a more complex tool for problem generation. We wanted to evaluate our schedules under different amounts of CPU load, therefore we assume the changing usage as the biggest difference between our problem instances (we refer to instances by their usage, e.g. USAGE 70). Moreover, we used different seeds in each case, providing a better variety of our experiments, as shown in Section 7.1.1.

In our initial experiments, we observed that benchmarks in our schedules performed a lower amount of iterations by up to 2% than SCHED_DEADLINE. We attribute this behaviour to DEmOS overhead, which is larger than the overhead of SCHED_DEADLINE. The increased overhead of DEmOS is not a surprise. First of all, DEmOS is a rather robust and complex program when compared to SCHED_DEADLINE. Moreover, SCHED_DEADLINE as a part of the Linux kernel is highly optimized, when compared to a user-space tool DEmOS. While some difference is expected, due to the unpredictability of OS operations, we wanted to compensate for this difference. Therefore, we increased the scheduled amount of work for DEmOS by 2 % (efficiency compensation variable set to 102 %). We show test case-specific settings in Section 7.1.1. For idea, we also show the number of scheduled tasks;

however, we can influence the task count only indirectly (e.g. by setting lower maximum runtimes).

Test case	1	2	3	4	5
Usage	50 %	60 %	70 %	80 %	90 %
Seed	100	101	102	103	104
No. of tasks	9	13	14	19	24

The rest of the values were left the same for all problem instances:

- Period (`--period`): 10 000 [ms]
- Maximum W_τ (`--maximum`): 5 000 [ms]
- Minimum W_τ (`--minimum`): 500 [ms]
- Task report (`--task_duration`): 500 [ms]
- Efficiency compensation (`--efficiency`): 102 [%]

Our tool creates not only schedules, as a DEmOS YAML configuration, but also bash scripts that execute benchmarks under SCHED_DEADLINE. Both can be found in the attached files.

Because SCHED_DEADLINE does not support CPU core/cluster affinity by default, the Linux CPU groups (cgroups) mechanism has to be utilized. We created a separate CPU group for each cluster.

7.1.2 Results

Each experiment was executed for 1200 seconds (20 minutes). This rather high experiment run is caused by heat distribution on the hardware. The iMX8 board is equipped with a heat sink on top of the processor, which prolongs achieving steady-state temperature significantly. We repeated each experiment 3 times to present accurate values.

Test execution schema

We present Gantt charts of our schedule and SCHED_DEADLINE in Figure 7.1 and Figure 7.2, respectively. Gantt charts in both cases capture the detail of one period (10 seconds). Both charts were created via trace-cmd Linux command-line tool and later exported in the kernelshark program. Gantt charts for the rest of the instances are listed in Appendix A.1. It is apparent, that our schedule utilizes the available bandwidth better, by spreading tasks over longer periods, leaving the CPU less often idled. Prolonging task execution period allows for usage of lower frequency, which is less power demanding.

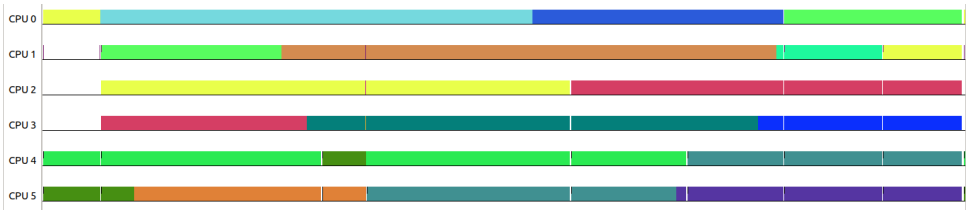


Figure 7.1: Gantt chart of one BEC window (10 000 ms) of DEmOS schedule for problem problem instance USAGE 70

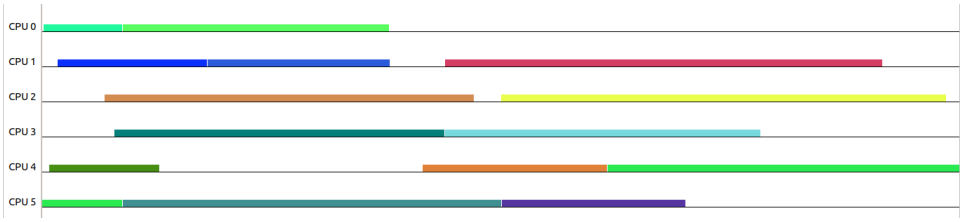


Figure 7.2: Gantt chart of one BEC window (10 000 ms) of SCHED_DEADLINE schedule for problem instance USAGE 70

Temperature

Since our main focus is temperature reduction, we deem the measured temperature plot as the most demonstrative. In Figure 7.3 it can be clearly seen that the temperature of the processor was lowered significantly (by more than 5°C on average) at problem instance USAGE 70. Similar achievement can be observed on all usage settings. We also display measured ambient temperatures, to prove that the difference was not caused by a change of external conditions. Ambient temperatures for both experiments are almost identical and therefore they overlap in Figure 7.3.

Not only that average temperature reduction was achieved, but we can also clearly smaller temperature fluctuation. This is also a very good sign of more stable task execution. Note that thermal boundaries are commonly dependent on imminent temperature rather than average temperature. More temperature stable execution thus offers a better guarantee of respecting the thermal envelope. All temperature comparison plots for all bandwidth usages can be found in the Appendix A.2.

We list measured temperatures for all experiments in Table 7.1. Temperature averages were measured after 600 seconds warm-up (we can observe only a small temperature grow after the first 10 minutes of run in Figure 7.3).

Power consumption

We also measured the average power consumption of each solution for each task. We show power consumption behaviour for problem instance USAGE

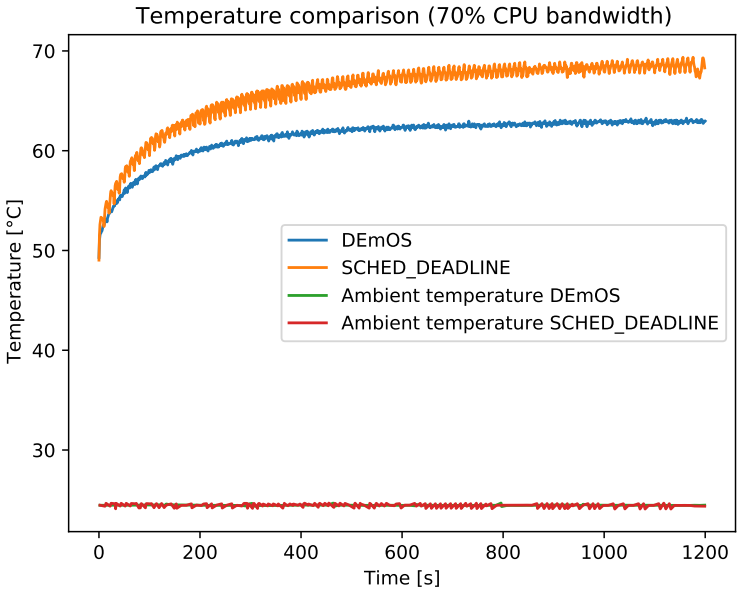


Figure 7.3: Temperature plot for problem instance USAGE 70.

Instance	DEmOS [°C]	SD [°C]
Usage 50	56.17 ± 0.27	60.04 ± 0.74
Usage 60	57.07 ± 0.25	62.13 ± 0.54
Usage 70	60.66 ± 0.28	65.79 ± 0.54
Usage 80	61.83 ± 0.38	65.34 ± 0.47
Usage 90	64.33 ± 0.51	65.66 ± 0.77

Table 7.1: Measured average temperatures

70 in Figure 7.4. Power consumption seems less stable than temperatures. This is mainly caused by two reasons. Temperatures may not raise or decline drastically in a short period of time due to the thermal capacity of the material. However, the power consumption of the processor is not limited in this way. Moreover, a different tool was used for power consumption data gathering. The external device which measures average power consumption is accessed through ssh protocol, which is more demanding than reading temperatures from the internal sensor on the chip. This prove to be an even bigger problem for thermobench, where despite high priority settings, data readings were sparse. Therefore, we created a specific script for ssh access only (power_measurement.sh), which was run under SCHED_DEADLINE to achieve equal priority settings.

Despite unstable measurements, we can still clearly observe higher average power consumption in all CPU bandwidth usage settings. All power consumption plots can be found in Appendix A.3. Again we put our measurements in numbers in Table 7.2. The difference in power consumption is also significant

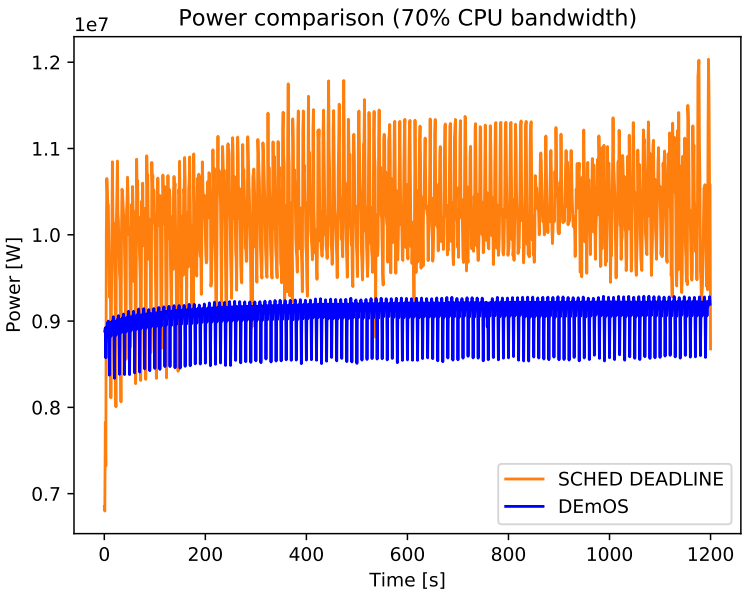


Figure 7.4: Power consumption plot for usage 70% of the total CPU bandwidth

Instance	DEmOS [W]	SD [W]
Usage 50	8.04 ± 0.55	8.9 ± 1.0
Usage 60	8.21 ± 0.47	9.4 ± 1.0
Usage 70	9.12 ± 0.22	10.31 ± 0.8
Usage 80	9.33 ± 0.4	10.1 ± 0.7
Usage 90	10.25 ± 0.38	10.6 ± 0.75

Table 7.2: Measured average power consumption

(almost 1.2 W on average in case of problem instance USAGE 70). In Figure 7.4 we can observe that not only temperature of the processor is dependent on the power consumption, but also the power consumption is influenced by temperature (even though this dependency is much weaker). Therefore, all averages were measured after 600 seconds of the warm-up period.

Work comparison

Finally, to prove that our scheduler achieved comparable work efficiency, we compare the workload executed. Once more, we present the result of the experiment conducted on problem instance USAGE 70 in Figure 7.5. From the plot, we can observe only minor differences in tasks iterations, where most of them are in our favour (meaning that experiments under our scheduler executed more work and still achieved better thermal and power consumption properties). Work comparison for all experiments can be found in Appendix A.4. One iteration (Y-axis) is an equivalent of 500ms runtime

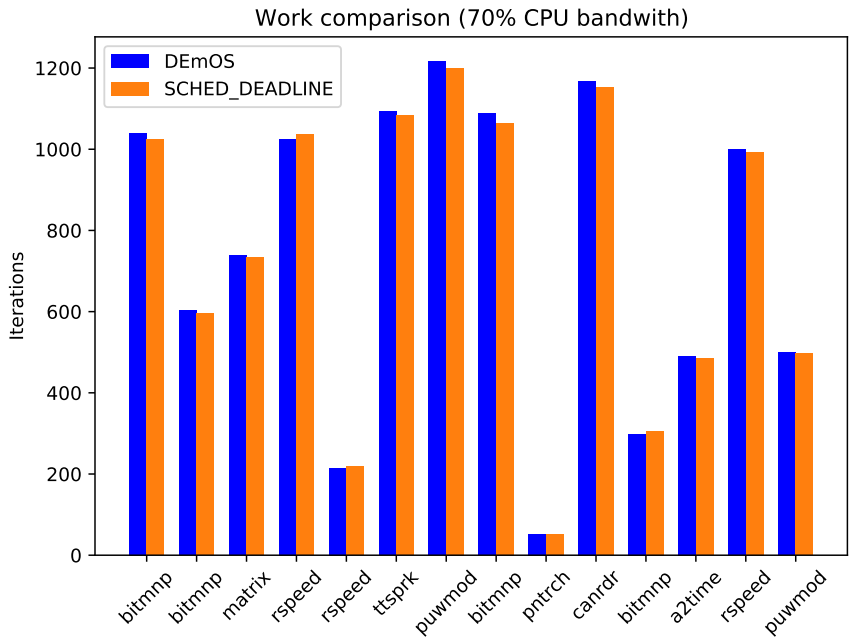


Figure 7.5: Work comparison for usage 70% of the total CPU bandwidth

on the highest available frequency. We put a comparison of all settings in numbers in Table 7.3. The difference in executed work is negligible and mostly in favour of our scheduler.

Bandwidth usage	50 %	60 %	70 %	80 %	90 %
Total iterations DEmOS	7550	8117	10526	11342	12798
Total iterations S_D	7496	8039	10439	11274	12834
Diff absolute	54	78	87	68	-36
Diff percentage	0.72 %	0.97 %	0.83 %	0.6 %	-0.28 %

Table 7.3: Work comparison

Power consumption estimation

As our scheduler solves linear equations, next to values of variables, it also finds the value of object function, which should be the upper bound of average power consumption. Because each cluster is solved separately, we also conducted separate tests for each cluster and measured average power consumption after an initial warm-up period of 600 seconds. We present our results together with estimates in Table 7.4

We can observe that our estimations are not accurate. This is caused by our $\lambda_{\tau,s}$ variables (see Section 5.2.2). We measured $\lambda_{\tau,s}$ on a cold processor (around 50 °C), therefore increased power consumption due to increased temperature is not accounted for.

Bandwidth usage	50 %	60 %	70 %	80 %	90 %
Cluster:	A53				
Estimated avg power [W]	6.06	6.32	6.44	6.67	7.11
Measured avg power [W]	6.38	6.78	6.93	7.17	7.69
Cluster:	A72				
Estimated avg power [W]	6.83	7.2	7.66	8.19	8.53
Measured avg power [W]	7.13	7.57	8.09	8.81	9.11

Table 7.4: Work comparison

7.2 Reactive scheduling

We initially evaluated our reactive policies on a firmly assigned policy level. This way we can compare the efficiency and granularity of each solution in a control environment. We show reactive behaviour in Section 7.2.2.

7.2.1 Static experiments

As previously mentioned, our policies execute the given schedule, until the reaction is needed (policy level increase). We chose a schedule from the previous experiment, problem instance USAGE 80. Higher CPU bandwidth usage leads to higher temperatures and thus proposed changes are more apparent. However, we decided not to opt for the highest possible setting due to measurement difficulties which are tight up with it, that is why we chose problem instance USAGE 80.

We proposed 3 different techniques for dynamic thermal management, frequency throttling, task skipping and iterative schedules. Due to the discrete, limited amount of available frequencies and cores, there is a limited granularity of the first two proposed techniques. There are 4 available frequencies, therefore we can lower the frequency only up to 3 times (we do not count level 0, where no change in a scheduled run is done). Our hardware offers 6 cores, therefore we assume the option of skipping up to 5 tasks at any time, to leave only one core running.

In contrast, our iterative schedule has unlimited granularity. Therefore, we create multiple iterative schedules, where each time we lowered the amount of executed work by 10 %. To prevent confusion, we note that we still use the initial problem instance USAGE 80, the same tasks in the same execution ratio are expected to be executed, but we schedule less work to each of them (e.g. iterative schedule of 50 % on problem instance USAGE 80 will create CPU load of 40 %).

We executed each firmly set policy separately for 1200 seconds. We measured averages after the first 600 seconds of the warm-up period and put

them into Table 7.5. We can observe a gradual lowering of temperatures. Testing and evaluation of iterative schedules were done in the same manner (firm setting, 1200 seconds, 600 seconds warm-up period).

Policy level:	1	2	3	4	5
Freq throttling [°C]	58.29	55.27	53.56	N/A	N/A
Task skipping [°C]	61.11	59.85	58.26	55.19	50.54

Iterative	20 %	30 %	40 %	50 %	60 %	70 %	80 %	90 %
Avg [°C]	48.08	50.22	52.95	54.16	55.33	56.78	59.46	60.64

Table 7.5: Temperature averages of reactive policies under firm level

To put results into context, we decided to compare frequency throttling level 2 with task skipping level 4 and iterative schedule of 60 % workload (highlighted green in Table 7.5 and Table 7.6), due to a very small difference in temperatures (0.25 % of a temperature difference between warmest and coldest experiment).

For illustration, we show temperature profiles in Figure 7.6, but since temperature averages are almost even, we mainly focus on work comparison in Figure 7.7. Iterative schedules follow the ideal work ratio, thus they comply with our definition of "fairness". We can see that frequency throttling is also following given work prescription, however more loosely.

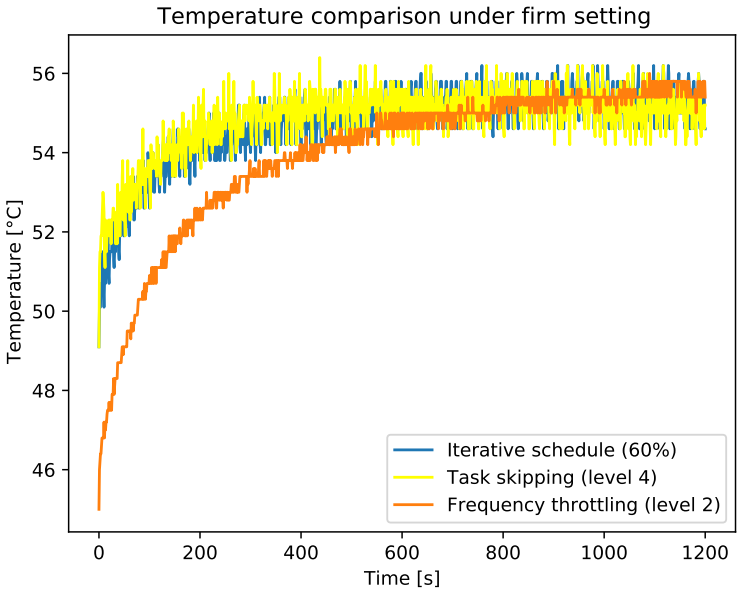


Figure 7.6: Temperature profiles of chosen reactive policies with firm settings

We put executed work into numbers to better represent our results in Table 7.6. We can see that if we compare previously plotted policies (frequency throttle level 2, task skipping level 4 and iterative schedule of 60% workload),

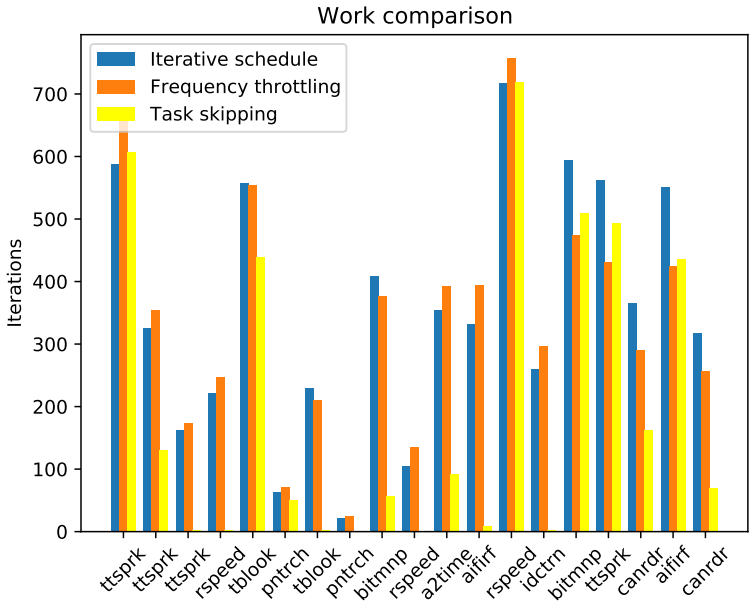


Figure 7.7: Work comparison of chosen reactive policies with firm settings

iterative schedule for the given setting beat frequency throttling by 207 iterations (3.07 %) and task skipping even by 2952 iterations (43.84 %).

Policy level:	1	2	3	4	5
Freq throttling [work done]	8790	6527	6319	N/A	N/A
Task skipping [work done]	9482	7619	5706	3782	1929

Iterative	20 %	30 %	40 %	50 %	60 %	70 %	80 %	90 %
Work done	2219	3354	4487	5618	6734	7837	8921	10013

Table 7.6: Work done by reactive policies under firm level

7.2.2 Reacting to external condition

We also conducted experiments, displaying the desired reactive behaviour. However, due to the current implementation of the DEmOS tool, we were only able to test our task skipping and frequency throttling policies. We set firm temperature bound and observed the behaviour of frequencies.

We can see an ideal case in Figure 7.8, where policy level was adjusted to ideal level after the first temperature bound overstepping, leading to temperature stable run. More violent policy level changes can be observed in Figure 7.9, where multiple levels of the policy were often changed in quick succession, leading to oscillating of the temperature. Vertical lines marks changes in policy levels.

Oscillating between policy levels is not ideal. In Figure 7.9 we can see that

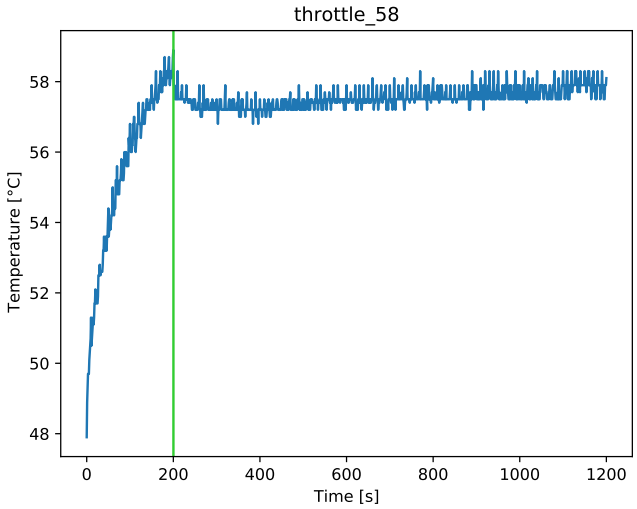


Figure 7.8: Frequency throttling policy with upper bound temperature of 58°C

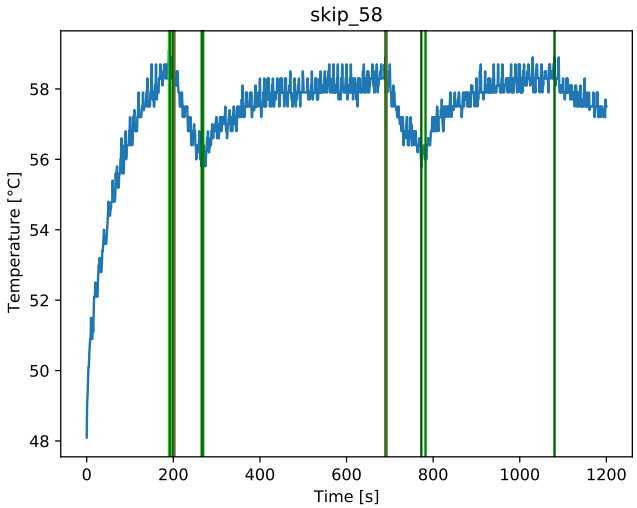


Figure 7.9: Task skipping policy with upper bound temperature of 58°C

after 200 seconds of run, the policy was set too strictly and therefore we lost part of CPU bandwidth. Choosing a correct policy level is not an easy task. Creating a mechanism, that reacts too dynamically will lead to loss of computational bandwidth. The reaction of too loose mechanism might not be adequate and policy could lead to system failure.

Chapter 8

Conclusion

In conclusion, we successfully created an offline best-effort task scheduler. Our scheduler is based on a linear model, which allows it to solve even big instances effectively. We implemented the scheduler for general usage and it should be portable to most processors.

Moreover, we experimentally evaluated the scheduler on a set of randomly generated instances, where we managed to achieve thermal and power consumption reduction, when compared to the state-of-the-art Linux scheduler SCHED_DEADLINE. Our scheduler consistently achieved better results and in some cases, it lowered the temperature of the processor by more than 8 % and power consumption by more than 12 % while it executed an even higher amount of iteration cycles on given tasks.

We also proposed and evaluated multiple reactive policies. We deem reactive policy using iterative schedules as the most effective and flexible; however, its dynamic behaviour is yet to be integrated with DEmOS. On static experiments, it outperformed other proposed policies under similar thermal conditions.

The second policy we proposed, implemented and evaluated even under dynamic conditions is the frequency scaling policy. We rate this policy as effective, even though its flexibility is limited by a number of available CPU frequencies.

The third proposed policy is the task skipping policy, which achieved greater flexibility, working in a wider range of temperature settings; however, the performance is rather poor when compared to other policies. Its flexibility is limited by the total amount of CPU cores.

8.1 Future work

Our model offers a great number of opportunities for future work, which are beyond the scope of this work. Mainly, our proposed reactive policy which utilizes precomputed iterative schedules promises good efficiency and flexibility, but its dynamic implementation would either lead to major changes in the DEmOS tool, or to a brand new tool for schedule execution.

Moreover, we made some assumptions when creating our model. We are confident, that some of them could be lifted; however, such implementation is beyond scope of this work. Nevertheless, we want to offer a possible approach.

In Section 3.3 we declared strict cluster affinity. However, our measurement suggests that scaling between clusters is very similar for all tasks. Therefore, if we would choose one cluster as a model, the efficiency of other cluster cores could be re-scaled by the introduction of another constant (e.g. that each core of the second faster cluster would be taken as 1.5 core of the first cluster, with different coefficient per each frequency comparison).

We also assume the start time and deadline are identical for all tasks. However, we can introduce different start times and deadlines by the introduction of multiple time frames, where each frame would consist of all "CPU settings" windows. This way, the model could still be linear and allow for usage of deadlines and start times; however, the complexity of the model would grow (addition of each new start time or deadline would increase the number of "CPU settings" windows by the total amount of "CPU settings"). This solution would also increase the amount of frequency switching.

Bibliography

[1] S. Banachowski, T. Bisson, and S.A. Brandt. “Integrating best-effort scheduling into a real-time system”. In: 25th IEEE International Real-Time Systems Symposium. 2004, pp. 139–150. doi:10.1109/REAL.2004.26.

[2] Andrea Bartolini et al. “Optimization and Controlled Systems: A Case Study on Thermal Aware Workload Dispatching”. In: vol. 1. July 2012.

[3] Andy Bavier, Larry Peterson, and David Mosberger. “BERT: A Scheduler for Best Effort and Realtime Tasks”. In: (Oct. 1998).

[4] Ondřej Benedikt et al. “Thermal-Aware Scheduling for MPSoC in the Avionics Domain: Tooling and Initial Results”. In: 2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). 2021, pp. 159–168. doi: 10.1109/RTCSA52859.2021.00026

[5] Jian-Jia Chen, Andreas Schranzhofer, and Lothar Thiele. “Energy minimization for periodic real-time tasks on heterogeneous processing units”. In: 2009 IEEE International Symposium on Parallel Distributed Processing. 2009, pp. 1–12. doi: 10.1109/IPDPS.2009.5161024.

[6] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Keith Whisnant. “Temperature Aware Task Scheduling in MPSoCs”. In: 2007 Design, Automation Test in Europe Conference Exhibition. 2007, pp. 1–6. doi: 10.1109/DATE.2007.364540.

[7] Vinay Hanumaiah, Sarma Vrudhula, and Karam S. Chatha. “Performance Optimal Online DVFS and Task Migration Techniques for Thermally Constrained Multi-Core Processors”. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 30.11 (2011), pp. 1677–1690. doi: 10.1109/TCAD.2011.2161308.

[8] David Hornof. “Offline scheduling of the safety-critical tasks within the isolation time-windows”. Diploma thesis. Czech Technical University, June 2021. url: https://dspace.cvut.cz/handle/10467/95363.

[9] E. Douglas Jensen. “Asynchronous Decentralized Realtime Computer Systems”. In: Real Time Computing. Ed. by Wolfgang A. Halang and Alexander D. Stoyenko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 347–371. isbn: 978-3-642-88049-0

[10] Almir Mutapcic et al. “Processor Speed Control With Thermal Constraints”. In: IEEE Transactions on Circuits and Systems I: Regular Papers 56.9 (2009), pp. 1994–2008. doi: 10.1109/TCSI.2008.2011589.

[11] Francesco Paterna et al. “Variability-Aware Task Allocation for Energy-Efficient Quality of Service Provisioning in Embedded Streaming Multimedia Applications”. In: IEEE Transactions on Computers 61.7 (2012), pp. 939–953. doi: 10.1109/TC.2011.127.

[12] Diego Puschini et al. “Temperature-Aware Distributed Run-Time Optimization on MP-SoC Using Game Theory”. In: 2008 IEEE Computer Society Annual Symposium on VLSI. 2008, pp. 375–380. doi: 10.1109/ISVLSI.2008.33.

[13] Andrea Rudi et al. “Optimum: Thermal-aware task allocation for heterogeneous many-core devices”. In: 2014 International Conference on High Performance Computing Simulation (HPCS). 2014, pp. 82–87. doi: 10.1109/HPCSim.2014.6903672.

[14] Jinggang Wang, B. Ravindran, and T. Martin. “A power-aware, best-effort real-time task scheduling algorithm”. In: Proceedings IEEE Workshop on Software Technologies for Future Embedded Systems. WSTFES 2003. 2003, pp. 21–28. doi: 10.1109/WSTFES.2003.1201354.

[15] Xiliang Zhong and Cheng-zhong Xu. “Energy-Aware Modeling and Scheduling for Dynamic Voltage Scaling with Statistical Real-Time Guarantee”. In: IEEE Transactions on Computers 56.3 (2007), pp. 358–372. doi: 10.1109/TC.2007.48.

[16] Junlong Zhou et al. “Thermal-Aware Task Scheduling for Energy Minimization in Heterogeneous Real-Time MPSoC Systems”. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35.8 (2016), pp. 1269–1282. doi: 10.1109/TCAD.2015.2501286.

[17] Claudio Scordino, Luca Abeni, and Juri Lelli. 2018. Energy-aware real-time scheduling in the Linux kernel. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC ’18). Association for Computing Machinery, New York, NY, USA, 601–608. DOI:https://doi.org/10.1145/3167132.3167198

[18] Mascitti, Agostino, et al. “An Adaptive, Utilization-Based Approach to Schedule Real-Time Tasks for ARM Big.LITTLE Architectures.” ACM SIGBED Review, vol. 17, no. 1, July 2020, pp. 18–23. DOI.org (Crossref), doi:10.1145/3412821.3412824

[19] M. Sojka, O. Benedikt, Z. Hanzálek and P. Zaykov, "Testbed for thermal and performance analysis in MPSoC systems," 2020 15th Conference on Computer Science and Information Systems (FedCSIS), 2020, pp. 683-692, doi: 10.15439/2020F174.

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), 2001, pp. 3-14, doi: 10.1109/WWC.2001.990739.

[21] ARM® White paper 2013, http://img.hexus.net/v2/press_releases/arm/big.LITTLE.Whitepaper.pdf [Online] Accessed: 1 December 2021.

[22] DEmOS repository. https://github.com/CTU-IIG/demos-sched [Online] Accessed: 30 December 2021.

[23] Thermobench repository https://github.com/CTU-IIG/thermobench [Online] Accessed: 30 December 2021.

[24] Embedded Microprocessor Benchmark Consortium official website. https://www.eembc.org/autobench/ [Online] Accessed: 16 December 2021

[25] The Linux man-pages project. https://man7.org/Linux/man-pages/man1/chrt.1.html [Online] Accessed: 16 December 2021.

[26] The Linux man-pages project. https://man7.org/Linux/man-pages/man1/taskset.1.html [Online] Accessed: 16 December 2021.

[27] The Linux man-pages project. https://man7.org/Linux/man-pages/man7/cpuset.7.html [Online] Accessed: 16 December 2021.

[28] INA219 data sheet, product information and support. https://www.ti.com/product/INA219. [Online] Accessed: 16 December 2021.

[29] Rostedt, Steven. "Using SCHED_DEADLINE." Embedded Linux conference + Open IoT summit Europe, October 2016, https://www.youtube.com/watch?v=TDR-rgWopgM [Online] Accessed: 16 December 2021.

Appendices

Appendix A

Experiments

In Chapter 7 we show graphs and charts of only one selected instance. Here we list all instances for comparison. Gantt charts of run in Appendix A.1 contain 3 periods.

A.1 Proactive scheduling - Gantt charts

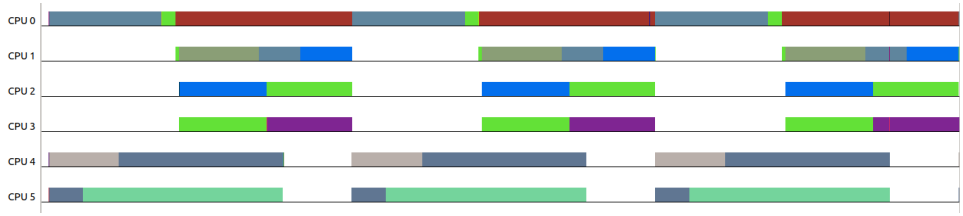


Figure A.1: DEmOS schedule of 3 BEC windows (30s runtime) of problem instance USAGE 50

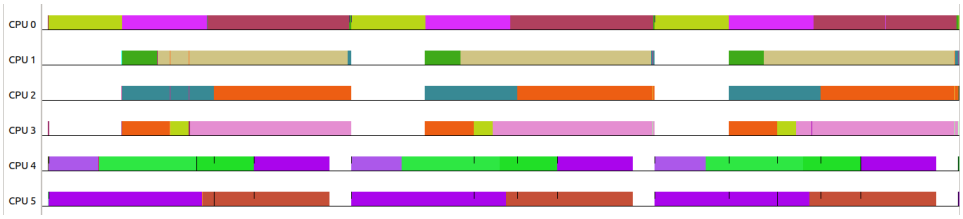


Figure A.2: DEmOS schedule of 3 BEC windows (30s runtime) of problem instance USAGE 60

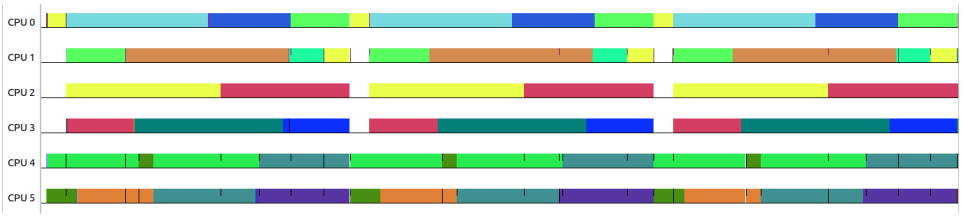


Figure A.3: DEmOS schedule of 3 BEC windows (30s runtime) of problem instance USAGE 70

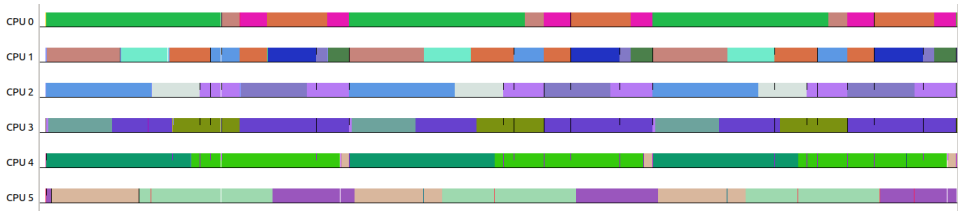


Figure A.4: DEmOS schedule of 3 BEC windows (30s runtime) of problem instance USAGE 80

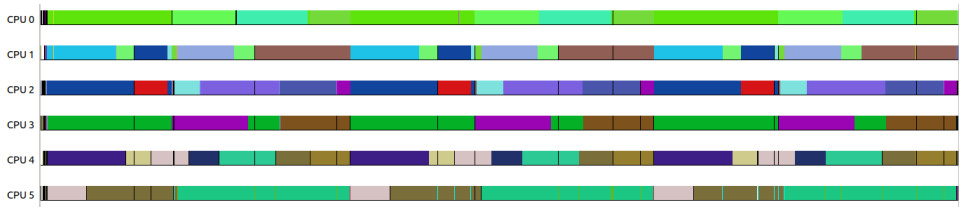


Figure A.5: DEmOS schedule of 3 BEC windows (30s runtime) of problem instance USAGE 90

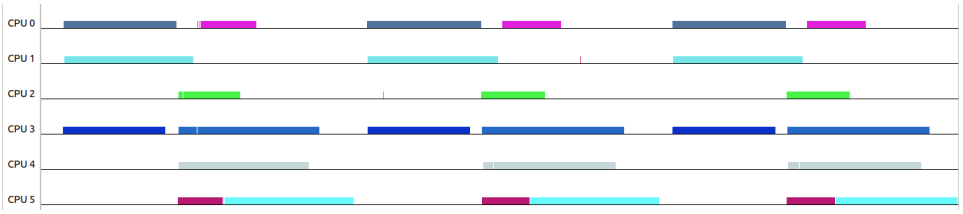


Figure A.6: SCHED_DEADLINE execution of 3 BEC windows (30s runtime) of problem instance USAGE 50

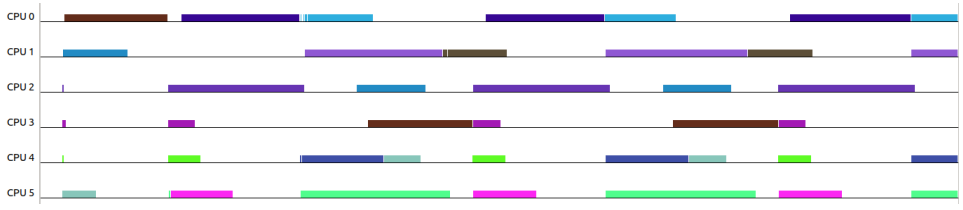


Figure A.7: SCHED_DEADLINE execution of 3 BEC windows (30s runtime) of problem instance USAGE 60

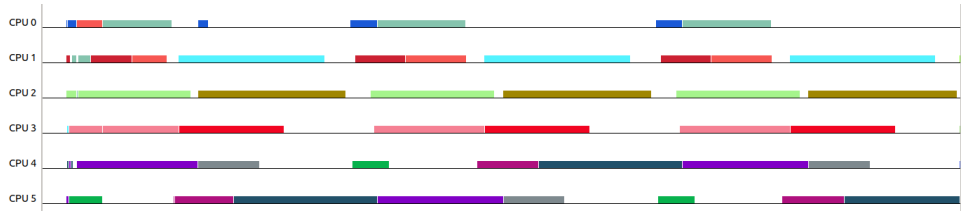


Figure A.8: SCHED_DEADLINE execution of 3 BEC windows (30s runtime) of problem instance USAGE 70

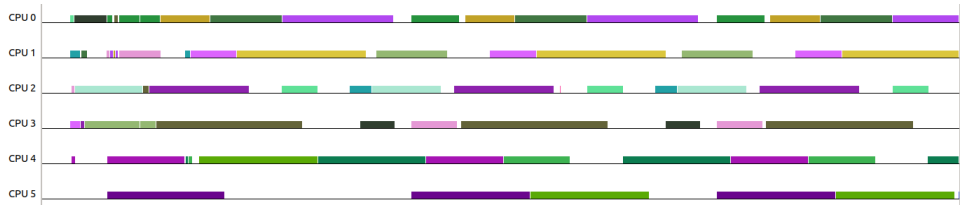


Figure A.9: SCHED_DEADLINE execution of 3 BEC windows (30s runtime) of problem instance USAGE 80

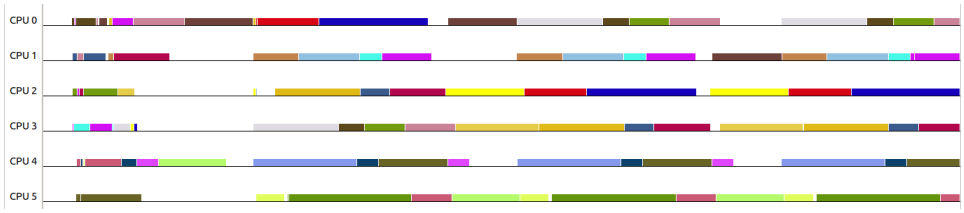
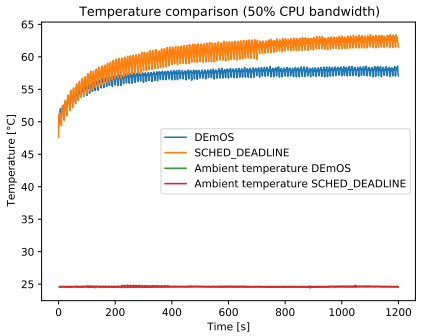
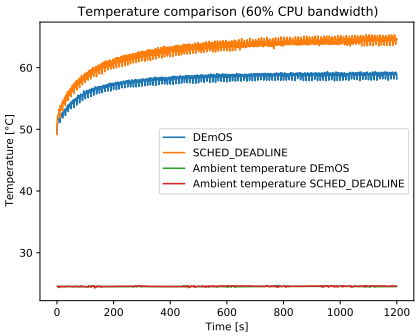
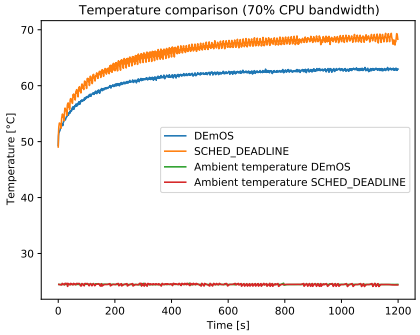


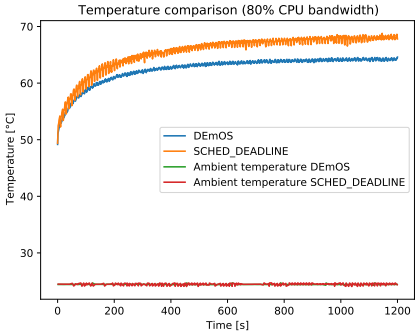
Figure A.10: SCHED_DEADLINE execution of 3 BEC windows (30s runtime) of problem instance USAGE 90

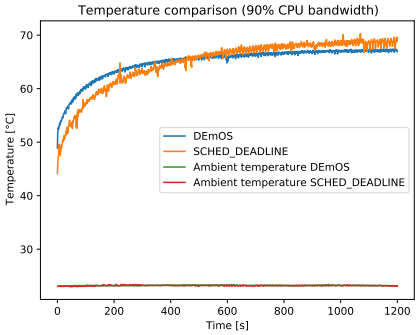
A.2 Proactive scheduling - temperature measurements



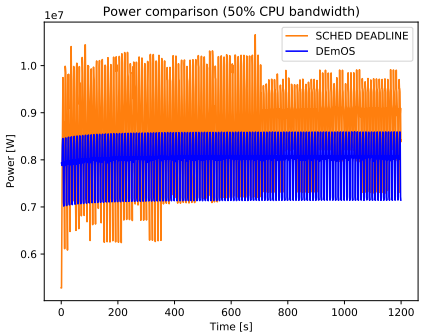


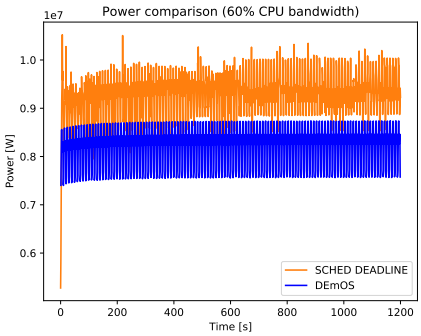


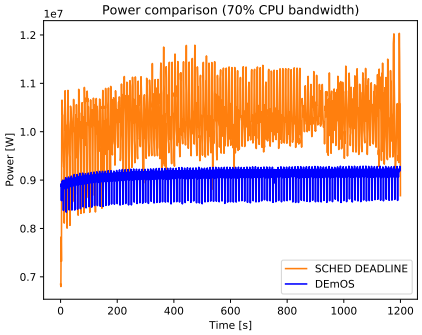


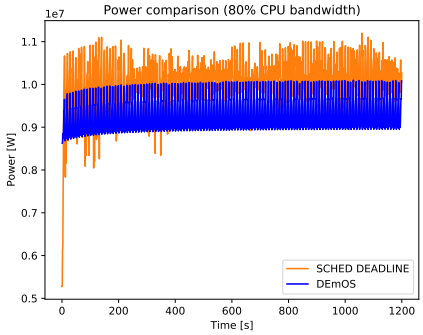


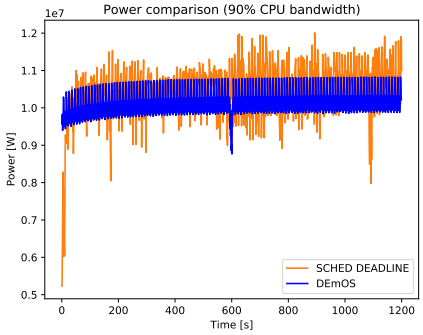
A.3 Proactive scheduling - power consumption measurements



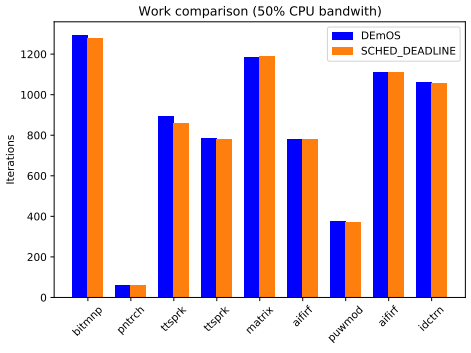


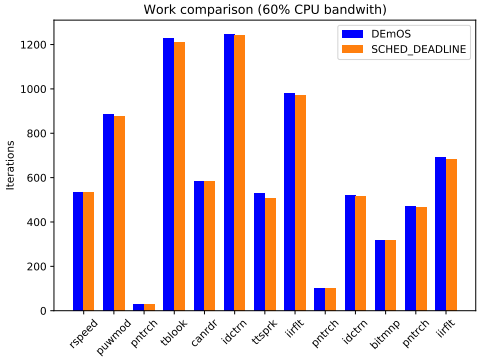


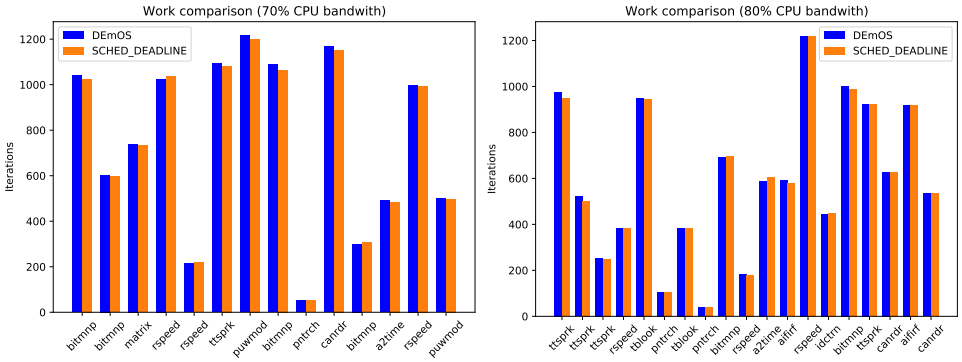


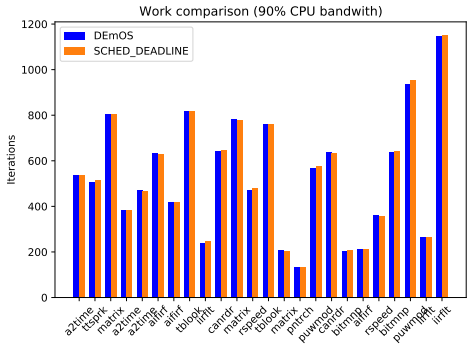


A.4 Proactive scheduling - Work comparison






Work comparison (70% CPU bandwith)
DEmOS
SCHED_DEADLINE
Iterations
0
200
400
600
800
1000
1200
bitmmp
bitmmp
matrix
rspeed
rspeed
ttsprk
puwmod
bitmmp
pntrch
canrdr
bitmmp
a2time
rspeed
puwmod
Work comparison (80% CPU bandwith)
DEmOS
SCHED_DEADLINE
Iterations
ttsprk
ttsprk
ttsprk
rspeed
tblook
pntrch
tblook
pntrch
bitmmp
rspeed
a2time
aifirf
rspeed
idctrn
bitmmp
ttsprk
canrdr
aifirf
canrdr


Work comparison (90% CPU bandwith)
DEmOS
SCHED_DEADLINE
Iterations
0
200
400
600
800
1000
1200
a2time
ttsprk
matrix
a2time
a2time
aifirf
aifirf
tblook
iirflt
canrdr
matrix
rspeed
tblook
matrix
pntrch
puwmod
canrdr
bitmmp
aifirf
rspeed
bitmmp
puwmod
iirflt
iirflt