DREUS: Precomputed Workload Splitting for Tunable Performance-Energy Balancing in Online Heterogeneous Multicore Scheduling

Merijn Bras (s3978605)

Supervisors:
Kristian Rietveld & Floske Spieksma

Bachelor Thesis– Wiskunde & Informatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

August 15, 2026

Abstract

Linux introduced Energy Aware Scheduling in 2019, to minimize energy costs while maintaining sufficient performance for CPU architectures containing different types of cores. It, however, enforces a singular operating point for energy efficiency and performance. In this thesis, we will introduce DREUS, an alternative dispatcher for task wake-ups that does not rely on a fixed trade-off between energy and performance, but instead exposes a family of operating points that are Pareto-optimal with respect to DREUS's energy model, through a single scalar σ_μ that determines a performance constraint. Given such a constraint, DREUS computes an optimal split of utilization and clock frequency between cores in an offline environment that minimizes energy while respecting the constraint, before deploying it online. We test DREUS with 4 different values of σ_μ, and find that tightening the constraint monotonically decreases sojourn time of tasks. All configurations tested improved on sojourn time, but consumed more energy than EAS. The strictest constraint that was tested can improve the average sojourn time of tasks by 20.8% with respect to EAS, while consuming 13.9% more energy.

Contents

1 Introduction

Ever since the surge in popularity of mobile phones and laptops, there has been a huge focus on energy efficiency in the software and hardware industry. Whereas a desktop PC is constantly plugged in, and can thus provide a lot of performance without worrying about battery drainage, a phone or laptop has to be much more delicate in balancing the two.
One of the most renowned developments of the last decade regarding this has been *Heterogeneous Multicore Processing*, often abbreviated as HMP. These are systems that use multiple kinds of processors/cores on the same SoC (System-on-Chip), with some being energy efficient, but slower (often referred to as E-cores), and others being more performant, but using more energy in the process (often referred to as P-cores). Here a group of identical cores (like all E-cores) is called a *performance domain*. HMP became popular with the introduction of ARM's big.LITTLE architectures in 2011 [44], and has since then been picked up by other CPU manufacturers like Intel (since Raptor Lake [39]).

A natural complexity that arises in HMP regards scheduling tasks in an energy-efficient way, while still keeping the system performant. When a process wakes up and needs to be assigned to a core to run on, the so-called *scheduler* needs to consistently make that choice in a short amount of time, in a manner that saves as much energy as is viable.
Many modern systems pair HMP with a technique called DVFS (Dynamic Voltage and Frequency Scaling), which allows the CPU to run at different clock frequencies / voltages at different points in time, where a higher frequency indicates higher performance[1], based on the workload on the CPU at that point in time. The software that dictates these frequencies is called a *frequency governor*.

In Operational Research this problem is called "dispatching in a heterogeneous parallel queue system" [8, 20], where the task is to efficiently assign incoming customers to one of multiple queues (see Figure 1) with each queue having its own policy of dealing with customers (e.g. FIFO). There is an important distinction to make between *online* and *offline* dispatching. In the offline case the set of incoming tasks and their arrival time is known *a priori*. For online dispatching this is not the case: tasks arrive at unknown times and the dispatch policy has to decide in the moment to which queue the task has to be assigned.
In the context of standard scheduling for interactive systems like phones or laptops with HMP we deal with tasks arriving at runtime. This means the scheduler has no insight into when tasks arrive, how much processing power the task needs and when it will sleep again until after the fact: thus the scheduling algorithm must use an online dispatch policy.

One of the current solutions for implementing a dispatch policy for HMP scheduling is Linux's EAS (Energy Aware Scheduling), introduced in Linux kernel version 5.0 to streamline HMP support [15]. It works together with the schedutil governor to pick a core to place a task that is waking up, comparing the energy cost of placing the task on an E-core versus placing it on a P-core, and choosing the core with the lower cost. Test cases showed EAS providing a 20% decrease in energy consumed by the CPU, varying based on the size of the workload [36].
An important property of EAS is that it is inflexible: EAS is turned on, or off. There is no way

[1] Generally approximated to scale linearly, so half the frequency corresponds to double the execution time, even though this approach is flawed (e.g. when processes are memory or I/O bound) [31]

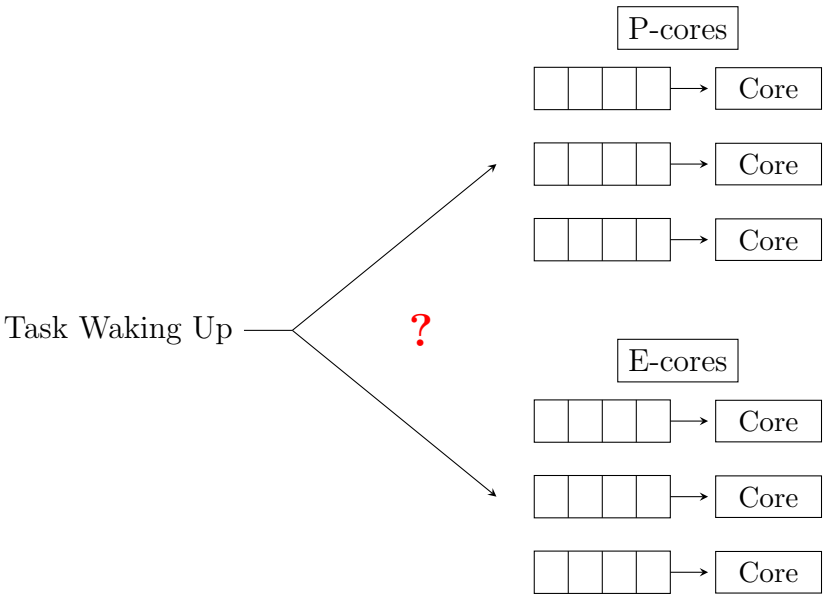


Figure 1: Dispatching waking up tasks (the customer) to one of the CPU's cores (the parallel queues).

to let EAS save half the energy, or double the energy. It provides just one balance point between energy consumption and performance.
When considering a concrete solution for this inflexibility, we have to note another property of EAS: the algorithm is greedy, so task placement decisions are only based on the state of the CPU at that point in time. As energy consumption is non-linear with respect to CPU utilization [21], and our notion of performance *is* linear with respect to utilization (as can be seen in Section 4.2.4), we have that for certain CPU utilization, a boost in performance leads to less extra energy consumption than for other CPU utilization. This means that if we want to maintain a certain performance standard with minimal energy consumption, we must globally choose where to bump that performance such that energy costs suffer the least.

In this thesis we propose "Dynamic Resource Efficient Utilization Splitting" (**DREUS**), to provide an alternative solution to HMP dispatching that is non-greedy and flexible in balancing performance and energy. This algorithm splits workload between E-cores and P-cores and exploits DVFS in such a way that average energy consumption is minimized, while also keeping average performance at a certain level (chosen beforehand), and never allowing performance to drop below a certain level. This is achieved by mathematically constructing an algorithm that takes into account all different possible amounts of workload to provide minimal energy consumption given the performance constraints. This construction is done in advance, before it is actually deployed in an online environment. In this sense DREUS creates a family of points that are Pareto-optimal with respect to our energy model and performance, and this provides us with the flexibility that EAS lacks: we can choose how we balance performance with energy by arbitrarily tightening the performance constraint.
The goal of this thesis will thus be to evaluate how different configurations of DREUS compare to EAS with respect to energy and performance in an online computing environment.

The rest of the thesis will now be structured as follows: Section 2 will first provide a theoretical background on the history and recent developments in HMP, and follow that up with a detailed explanation of the inner workings of EAS, including the way the Linux kernel tracks utilization of tasks and CPUs.
Section 3 presents an overview of related and recent literature regarding this topic to uncover the gap this research bridges in current academic literature.
Section 4 describes the detailed modeling of DREUS, and how it will be tested on actual hardware. The results of the experiments/tests will then be shown and discussed in Section 5, after which we finish the thesis with an overview of the essential conclusions and remarks on possible further research in Section 6.

2 Background

In this section we provide a brief history on the development of Heterogeneous Multicore Processors and how DVFS works in correspondence with them, and then go into more detail about Linux's scheduling and frequency governing algorithm that deals with HMP.

2.1 Heterogeneous Multicore Processors

As stated in the introduction, HMP primarily gained traction as an industry standard for mobile phone SoCs with ARM's big.LITTLE. Multiple major mobile SoC manufacturers have introduced heterogeneity in their processors since then. For example, Apple introduced it with their A10 SoC in 2016, and Qualcomm used it as far back as 2014 with the Snapdragon 810. Although both initially had their issues[2], the hardware technique has seen major development and improvements over time. At the time of writing, there are multiple mobile SoC manufacturers that use three different performance domains on the same CPU, where instead of P- and E-cores, cores are grouped under so-called Prime-, Power- and Efficiency-cores [38]. For the sake of simplicity we will only consider 2-tier heterogeneous SoCs (with only P- and E-cores) within DREUS, but the methodology is extensible to a system with any number of performance domains, although this comes with increased computational costs.

2.2 DVFS

In the Linux kernel, we use the term performance domains as sets of cores that scale together with respect to frequency [29]. On many systems, these also directly correspond to the different clusters of cores in the hardware (i.e. an E-core has to run at the same frequency as another E-core, but a P-core can differ from an E-core frequency-wise). This also explains the name *performance domain*: a group of cores that has to simultaneously run at a certain performance point.
We note that performance domains and core clusters do not always line up: for example, heterogeneous Intel CPUs always allow each logical processor's frequency to be controlled independently [45, `/drivers/cpufreq/intel_pstate.c`]. Another important thing to note about DVFS is that

[2]The Snapdragon 810 was notorious for overheating [18], and the Apple A10 only allowed for one performance domain to be active at a time [26].

frequencies for performance domains cannot be selected from a continuous range: there is a discrete set of *OPP*s (Operating Performance Points) on which the CPU is allowed to run. The number of points is hardware-dependent; on the Intel Raptor Lake machine which will later be used for experiments, a P-core has 59 of these points, and an E-core has 37.
Now importantly, we have that a core's power consumption P relates to voltage V and frequency f like $P \propto V^2 \cdot f$ [42], and V relates to f in a superlinear fashion that is dependent on the hardware [37]. This non-linearity is the basis for the energy model we provide in Section 5.2.

For most CPUs, the OPP at a certain point in time is determined by the OS, which bases its decision on the current CPU utilization of the cores in question (more on this in Section 2.3.3). This is again not the case for all CPUs: Since the 6th generation of Intel processors, a concept called hardware P-states (HWP) is used. In this context the frequency level a performance domain runs on is mainly decided by the processor itself, not the currently running frequency governor. The operating system can still request a certain frequency, but in reality the hardware makes the final call, and the actual frequencies the domain runs at can thus differ from what the operating system asks for [48].
Intel motivates this by stating that the factors operating systems take into account for their decision are simply insufficient for choosing the performance state of a domain, with the hardware taking more metrics into account, such as the thermal limits of the core [3, 17].
This makes it important to note a certain bias that occurs, when evaluating a scheduler that bases its actions on the active frequency governor if run on such a CPU architecture: as the requests of the frequency governor do not properly reflect the actual OPPs for the CPU's cores, the decisions of the scheduler might be based on false assumptions, leading to inefficient scheduling.

2.3 The Linux Scheduler

Now that we have a general overview of the way HMP behaves, we can focus on the way the Linux kernel deals with HMP to provide energy-efficient task placement at runtime.
We start by looking briefly at the general scheduler for Linux, and then take a general look at the frequency governors integrated into Linux. Afterwards we take a deeper dive into the inner workings of schedutil, to then finally be able to give a thorough insight into the policy behind EAS.

2.3.1 EEVDF

We start by considering the standard Linux scheduler, which is called EEVDF (Earliest Eligible Virtual Deadline First), implemented in 2023 by Peter Zijlstra as the successor to CFS (Completely Fair Scheduler) [16].
For our purposes it is most important to note that both EEVDF and CFS have roughly the same goal: utilize all cores equally at all times. When a task wakes up, EEVDF tries to place it on an idle CPU (if there is one). Furthermore, workloads between cores are periodically balanced to split them evenly between cores (see kernel function `sched_balance_domains`), and whenever a core is going idle, it starts to aggressively pull tasks from surrounding cores (`sched_balance_newidle`) [45, `/kernel/sched/fair.c`].

When a task gets assigned to a CPU, it gets placed on that CPU's *runqueue*, along with all other tasks that are assigned to that CPU. EEVDF tries to divide available CPU time equally between every runnable process in its runqueue (just like CFS, which is where the name *Completely Fair* came from) [3] [34].

It is important to note that in practice, EAS is a part of EEVDF. When requirements are met (i.e. the system is HMP capable), EAS is enabled and does two things: it takes over task placement for waking up tasks, and disables the prior discussed periodic load balancing (reasoning on why this is done, and why it is even viable is discussed in Section 2.3.4). The per-core runqueue handling by EEVDF is not modified, and thus will also not be the focus of this thesis.

2.3.2 Frequency Governors

The policy that EAS uses is highly dependent on the inner workings of the frequency governor schedutil. More specifically, if schedutil is not the active frequency governor, EAS will also be disabled.
As such, it is worth the effort to dive deeper into the policy behind Linux's frequency governors, and the gap that schedutil fills with respect to the other governors. Apart from schedutil, Linux has the following integrated governors [9]:

Performance This governor sets all performance domains to simply run at the highest available frequency at all times.

Powersave The opposite of `Performance`, letting all performance domains run at the lowest available frequency at all times.

Userspace Lets the user determine the frequency by writing the desired frequency to the file `/sys/devices/system/cpu/cpu*/cpufreq/scaling_setspeed` (if supported).

Ondemand This governor bases the choice of the frequency on the load of the corresponding performance domain. After every time window (typically around 10 ms or more) it determines CPU usage and sets its frequency purely based upon that information.

Conservative Behaves approximately the same as ondemand, but instead of immediately letting the CPU jump to the frequency corresponding to a certain CPU usage, it only takes a step into the direction of that frequency. So instead of only taking into account CPU usage, it also bases its actions on the current frequency.

Although there certainly is a precedent in letting the chosen frequency be based on CPU usage, the ondemand governor has two problems: as CPU usage is bursty for interactive workloads, it can result in aggressive frequency jumping, which in turn can lead to higher battery drainage [9].
Secondly, for the same reason, ondemand always lags behind the actual performance a performance domain needs: the frequency is changed based on what the CPU needed *last* window.

[3] This assumes that every task in the runqueue has the same priority, also referred to as *niceness* (note that a lower niceness corresponds to a higher priority). If a task has a higher priority, the reported time it spent on the CPU gets scaled down, so that it has to run longer on the CPU to get the same reported runtime as other tasks. This concept is called *virtual runtime*.

The new window during which the cores actually run at the modified frequency might benefit from another frequency entirely.
The conservative governor solves the first issue by making frequency changes less rampant, but in the process makes the governor lag behind even more, as it may take multiple time windows for the governor to catch up to a certain needed performance.

To combat this, the governor needs a way to accurately *predict* what the CPU load will be in the next time window, and base its decision making on that prediction. This is exactly what schedutil offers as a frequency governor. In the next subsection we will go into more depth about how schedutil achieves this.

2.3.3 Schedutil

Schedutil achieves accurate load prediction of CPUs by providing each core with a *capacity* score that represents the workload it can handle in a certain amount of time. It is normalized from 0 to 1024, with a P-core operating at its maximum frequency having a capacity of 1024. Capacity scales linearly with computing power, so a P-core with a capacity of 1024 can handle twice as much load as an E-core with a capacity of 512. In a Linux environment, the maximum capacity of a core "*" is documented at the file `/sys/devices/system/cpu/cpu*/cpu_capacity`. The capacity of a core is also influenced by the frequency it is running on[4]. As mentioned earlier, frequency is assumed to scale linearly with capacity, so if a core is running at half its maximum frequency, it reports half its maximum capacity.

Now, to be able to predict the workload of a performance domain, we shift the load tracking away from cores, and focus specifically on the average load over time per individual task. This concept is called PELT (Per Entity Load Tracking).
Each task gets assigned a *util* (short for utilization) score, referred to as `util_avg` in the Linux kernel. We look at time windows of 1024 μs, and keep track of how many of those μs the task was busy on its assigned core, resulting in a score between 0 and 1024. We then scale this score by the capacity of the core it ran on. So a task running 100% of the time on a core with a capacity of 366, would get a score of 366, and a task running 50% of the time on a core with max capacity (1024), would get a score of 512.
However, instead of only taking into account the score of the last time window, it uses an EWMA (Exponentially Weighted Moving Average) to calculate the util score of a task t. That is, we define a row $u = (u_0, u_1, u_2, \cdots) \in [0, 1024]_{\mathbb{N}}$, where u_n corresponds to the score that t got n time windows ago, and get:

$$\mathrm{EWMA}_t(u) \;=\; \sum_{i=0}^{\infty} u_i \cdot y^i, \qquad \text{with } y^{32} = \frac{1}{2}.$$

Because we set $y^{32} = 1/2$, we have that a score's influence on the EWMA halves every 32 time windows (~ 32 ms). We now scale EWMA_t back to a range of 0 to 1024 to get the final util score of our task t:

$$\mathrm{Util}_t(u) = \frac{\mathrm{EWMA}_t(u)}{\mathrm{EWMA}_t(1)} = \frac{\mathrm{EWMA}_t(u)}{\sum_{i=0}^{\infty} y^i} = \frac{\mathrm{EWMA}_t(u)}{1/(1-y)}.$$

[4]This is not entirely true, when scaling to account for capacity, the kernel first scales for capacity, and then scales again afterwards for frequency. But combining the two is equivalent and more elegant for our purposes.

We can easily see that this indeed ranges from 0 to 1024, by taking $u_{\max} = (1024, 1024, \cdots)$ and calculating that:

$$\text{Util}_t(u_{\max}) = \frac{\sum_{i=0}^{\infty} 1024y^i}{\sum_{i=0}^{\infty} y^i} = 1024.$$

By taking this moving average, we take into account how much t kept its assigned core busy in the past, such that we get a more accurate depiction of how much t will keep the core busy in the future. It is less sensitive to bursty behavior: for Util_t to significantly rise, the task must consistently keep the core busy for a longer period of time.

Because u_n gets scaled by the capacity of the core it is running on, it also becomes invariant to task migration: if t (assuming the computing time it needs is invariant of time) accumulated a util score of 200 on core A, it will also accumulate a util score of 200 on core B[5].
Because of this, we can look at the capacity of a core as the amount of util it can process in a time window: if we load a few tasks with a total util score of 600 on a core with a capacity of 1000, it will be busy 60% of the time. In this manner we can also define a util score for a core itself: the total sum of util scores of tasks running on that core.

Now this all finally ties into how schedutil decides the frequency of a performance domain. Every OPP of a core has a corresponding capacity the core has for that clock frequency, namely $\text{Cap} = \frac{\text{Freq\_curr}}{\text{Freq\_max}} \cdot \text{Cap}_{\max}$.
Schedutil now looks at the highest util score of all cores in its domain, and then chooses the lowest OPP that corresponds to a capacity higher than that Util score.
In other words: it chooses the lowest frequency such that all cores in the domain are able to deal with their assigned util.
To account for sudden spikes in CPU usage, schedutil accounts for a 25% overhead (as apparent in the kernel function `sugov_effective_cpu_perf`). So if we let $c_0, c_1, \ldots \in C$ be the cores in a performance domain, Util_{c_n} be their util scores, F be the set of available frequencies and $\text{Cap}(f)$ be the capacity associated with a frequency f, then the frequency f^* that schedutil will choose is determined as follows:

$$\mathcal{F}^* = \{f \in F \mid \text{Cap}(f) > 1.25 \cdot \max_{0 \le i \le n} \text{Util}_{c_i}\},$$

$$f^* = \begin{cases} \min \mathcal{F}^*, & \mathcal{F}^* \neq \emptyset \\ f_{\max}, & \text{otherwise} \end{cases}.$$

2.3.4 EAS

Now we have all the necessary insight to discuss EAS in detail. When a task wakes up, the kernel function `select_task_rq_fair` is called, which returns the core in whose runqueue the task will be placed. One of the first things this function does is checking if EAS is enabled, which is dependent on a few things [30]:

- The CPU must use HMP. If the processor is symmetric (i.e. all cores are of the same type), EAS does not provide significant savings in energy.

[5]You can make sense of this by seeing that if core A has twice the capacity of core B, it will have twice the computing power, and thus will have dealt with t in half the time.

- EAS uses an Energy Model (EM), that contains a function `em_cpu_energy` which calculates (heuristically) the energy consumption of a performance domain. It takes in the maximum util score present in the domain and finds the OPP that schedutil would choose. It then queries the energy this OPP costs to run on from a lookup table . This lookup table is populated from the callback `active_power` [45, `/kernel/sched/energy_model.c`]. The callback must be present and given by the hardware manufacturer. Otherwise EAS cannot calculate energy costs.

- As the energy model makes its energy predictions based on schedutil, that must also be the chosen frequency governor.

- If the system in question is 32-bit, there is a risk of overflowing when estimating energy, so in that case there is a limit on the number of cores that can be present in the system, namely 16. If the system is 64-bit this problem vanishes, and the limit gets as high as 4096 [45, `/kernel/sched/energy_model.h`].

- On a last small note, EAS is also disabled on CPUs with simultaneous multithreading (SMT), as EAS considers threads as independent cores, which can actually reduce energy efficiency.

If EAS *is* enabled, it takes over dispatching tasks when they wake up, under the important condition that no single core is *overutilized*. Before going into detail about what this concretely denotes, it is first useful to note a few things about what it even means for a task to "wake up".

Every now and then, a task needs to wait for certain data/input. For example, when a task needs to wait for a certain user input (I/O) or needs data from memory that is not loaded in RAM (also known as a page fault), it *blocks*. The task is not recognized anymore as runnable and gives way for other tasks to run on its assigned core.
The moment the task unblocks again and becomes runnable is the exact moment we define as "waking up", and this is when `select_task_rq_fair` reconsiders on which core to schedule the task.
It is important to note that when a task never blocks, that also means it in theory needs infinite CPU time at each point in time: it can always stay computing indefinitely.
Let us say we have a program that needs to accurately compute the value of π^2:

```
double i = 1;
double pi2 = 0;

while(1){
    pi2 += 6/(i*i);
    i++;
}
```

Unlike a program that blocks every now and then, this program would fully utilize the core it runs on indefinitely, resulting in the corresponding core reporting a maximum util score as long as the task runs on that core.

Now we can turn again towards overutilization. We count a core as overutilized when it reports a util score of more than 80% of its maximum capacity (in kernel function `util_fits_cpu`,

`fits_capacity` is called [45, `/kernel/sched/fair.c`]).
When this happens, there is no spare capacity anymore for the 25% overhead for spikes in CPU usage (as $0.8 \cdot 1.25 = 1$)[6], and there is a significant risk that the core in question will hit maximum utilization. This would however break EAS, as EAS makes two important assumptions that break when cores hit maximum util [30]:

1. EAS uses the util scores from prior time windows to help decide on task placement. However, for util scores to be accurate it is necessary that all tasks on a core get their desired CPU time. If a core is fully utilized, the util scores get capped by the available CPU time and thus get underestimated. If for example a set of tasks with a total util score of 1000 gets placed on a core with a capacity of 500, after a while the total reported util of those tasks will also be capped at 500, even though their truly needed util is double of that.

2. We noted earlier that if EAS is enabled, periodic load balancing by EEVDF is disabled. As EEVDF does not consider energy efficiency, it would immediately undo the work EAS has done to save energy if the load at each core would frequently be rebalanced.
   The reason that EAS can afford to do this is because if no core is overutilized, it means that every task running on a core gets blocked fairly frequently: as we noted earlier, if one of the tasks would not block, the core it runs on would hit 100% utilization after a while. So only considering migration of tasks when they wake up is sufficient in practice, iff no core is overutilized.

To solve this, if a core *is* overutilized, EAS disables itself, and EEVDF takes over fully. It recognizes that because of the high workload, we momentarily stop caring about energy efficiency until EEVDF has reduced the util of all cores until they are under the overutilization threshold.

Finally, if EAS is enabled and no core is overutilized, `select_task_rq_fair` calls `find_energy_efficient_cpu`. This function does the following:
Let us say a task t wakes up and must be scheduled on a core in an energy-efficient manner. If possible, it first computes the energy cost of keeping t on the previous core it ran on as a baseline. EAS now looks at the two power domains, and finds the core with the lowest util for each. It then computes the energy cost of placing t at either of these cores.

Computing energy when placing t at a certain core is done as follows (kernel function `compute_energy` [45, `/kernel/sched/fair.c`]): the function calculates energy cost of a performance domain by first finding the core with the most util in that domain, and then the total util across all cores in the domain. It then passes that to the energy model with the function `em_cpu_energy`.
As mentioned earlier, the energy model looks at the OPP schedutil would choose based on the just calculated maximum util, and queries a lookup table that gives back the power it costs the domain to run at that OPP.
Now the return value for the estimated energy calculation becomes power$\cdot \frac{\text{sum_util}}{\text{cap}}$.

Finally EAS chooses to dispatch t to the core which returns the lowest energy cost. If it is a tie between one of the options and the baseline (keeping t at the previous core it ran on), it

[6]Because of this, another way of looking at overutilization, is that a core counts as overutilized when schedutil lets it run on its maximum frequency.

chooses the baseline because of cache locality[7]. A good concrete example can be found in [30] (although the example does not account for schedutil's 25% overhead when choosing OPPs for simplicity).

A good final thing to note is that because EAS is a greedy algorithm, the reported energy costs do not need to be accurate: we just need that if the heuristic used by EAS reports one state as using more energy than another, then the actual energy that state costs should also be higher than the other. Mathematically we state that if we have a function energy : $S \mapsto \mathbb{R}$ with S representing all possible CPU states, which maps a state to the actual energy that state consumes, and if we have the function energy_heuristic : $S \mapsto \mathbb{R}$, we need to have:

$$\text{energy\_heuristic}(s_1) > \text{energy\_heuristic}(s_2) \iff \text{energy}(s_1) > \text{energy}(s_2).$$

We say here that energy_heuristic and energy both induce the same ordering on S. If we have this, EAS still makes the same decisions every time, independent of how much the heuristic differs from actual energy costs: EAS only cares about the way two energy costs relate to each other, not by how much. As such, the heuristic used does not even necessarily need to return values in Watts, but can also be an abstract unit/scale. If we look at the callback[8] Intel provides for their CPU driver [45, `/drivers/cpufreq/intel_pstate.c`]:

```
static int hybrid_get_cost(struct device *dev, unsigned long freq,
                                    unsigned long *cost)
{
        /* Facilitate load balancing between CPUs of the same type. */
        *cost = freq;
        /*
         * Adjust the cost depending on CPU type.
         *
         * The idea is to start loading up LPE-cores before E-cores and start
         * to populate E-cores when LPE-cores are utilized above 60% of the
         * capacity.  Similarly, P-cores start to be populated when E-cores are
         * utilized above 60% of the capacity.
         */
        if (hybrid_get_cpu_type(dev->id) == INTEL_CPU_TYPE_ATOM) {
                if (hybrid_has_l3(dev->id)) /* E-core */
                        *cost += 1;
        } else { /* P-core */
                *cost += 2;
        }

        return 0;
}
```

[7]Because the task ran on this core earlier, there is a chance that data that t needs is still in its cache, improving performance and even a bit on energy efficiency [27].

[8]One could notice that the piece of code talks about cost instead of power, but in this context, cost is just power divided by the capacity of the core in question, such that the final calculation for `em_cpu_energy` discussed earlier simply becomes cost·sum_util.

As we can see this is quite a simple heuristic. As we discussed earlier, Intel uses Hardware P-states to control their frequencies. As such, the Intel driver treats "frequency" for the energy model as a value between 2 and 5, roughly correlating to respectively 40%, 60%, 80% and 100% of the full capacity (see `hybrid_active_power`). Now to get the energy cost it is simply the case that if the core is a P-core, you add 2 to that value, and in the case of an E-core you add 1[9].

Even if the callback provided by the hardware manufacturer returns values in Watts and is quite accurate, `em_cpu_energy` still assumes a linear relation between energy consumption and core utilization, which, as discussed in the introduction, can lead to the energy model becoming inaccurate anyway.
We, however, conclude that these models were built around preserving the ordering of CPU states with respect to the one induced by the actual energy, and as such still work well for EAS.
But, because of this, for a dispatch policy that actually needs to base its decision on the quantitative difference in energy between different states, the energy model is not sufficient. A dispatch policy made to keep down energy consumption globally by minimizing average energy cost over time instead of only locally (like EAS) has no use for the heuristic provided by Intel, as the policy returning the lowest average value returned by the heuristic, can be very different from the policy actually returning the lowest average energy cost.

3 Related Work

Recent literature regarding HMP has mostly been focused on scheduling, varying on the goals they try to achieve. In this section we will go over the different algorithms introduced by this literature to find where DREUS differs and the exact gap it fills.

There has been some effort in developing algorithms for HMP in offline environments (where the scheduler has a priori knowledge about the tasks at hand) [28, 49, 13, 12, 10, 14].
Here we have, in contrast to DREUS, that the set of tasks is predetermined, and as such the plausible state space of ways to schedule tasks grows enormously: there simply is more available information at a given point in time.

There is also a distinction in the goal different schedulers try to achieve: in [7] it is recognized that mobile devices (which are most likely to have HMP) are prone to overheating (as it is not viable to cool them with a fan), and it focuses on minimizing thermal violations, while also reducing energy costs.
Another target that schedulers may try to maximize is fairness (i.e. trying to provide equal resources to all tasks) [32, 23], Quality of Service (QoS) when talking about datacenters [19], or just plain performance [43].

When reducing the scope to online schedulers that try to minimize energy, like DREUS, a technique that is used in multiple schedulers involves giving each task a score on how efficient it will be to run that task on an E/P-core [40, 41], based on the energy cost and performance of that task on

[9]The code also refers to so-called LPE (Low Power Efficient) cores, which some Intel CPUs use to save energy in extremely low-power scenarios. They are not relevant to us.

both E- and P-cores. These scores are used to locally calculate the optimal task placement for energy efficiency and performance.
Another quite popular approach in recent years (at the time of writing) is taking advantage of deep/reinforcement learning to make scheduling decisions based on task and core information [25, 47, 4].

We note that the policies behind these schedulers fail by our criterion of a provable global energy cost optimality, either because the policy is greedy, or because it is produced by a reinforcement learning algorithm, which has the notable risk of falling into a local minimum instead[10].
For mathematical rigor, we would need to turn to Operational Research. As mentioned in Section 1, the related part of this field is often referred to as *online dispatching in a multi-queue system* (sometimes also referred to as a multi-*server* system). However, comparatively little of this literature focuses on heterogeneity [24], and those that do mostly cover "large scale" servers, "large" to such an extent that it is not realistic for arriving tasks to consider all servers if a low response time is desired [1, 2, 11].
A dispatching algorithm that comes quite close to filling the so-far described gap is in [33]. However, the algorithm in question is applied to dispatching incoming tasks to one of multiple servers in a cloud environment, and in the process it assumes tasks arrive according to a Poisson distribution, which is not considered accurate in a realistic end-user setting (as realistic workload is often more bursty than a Poisson process would suggest) [22].

Another important notion on these schedulers is that they do not solve EAS's inflexibility, and all still provide a singular operating point. In [35], a few of these operating points are provided, but it still does not give a wide range of flexibility.
We lastly consider [37], where an algorithm is proposed, called HEMP, that lays some groundwork for DREUS: it proposes an algorithm that mathematically deduces an optimal split between E- and P-cores and clock frequency in an offline environment for energy efficiency, with certain performance constraints, and then randomly assigns incoming tasks to cores with a probability distribution built from that optimal split. This is very much a non-greedy algorithm and takes energy consumption and performance into account on a global scale. It, however, is mainly focused on trading performance for energy in high-workload scenarios, whereas DREUS focuses on varying interactive workloads. The gap that DREUS also covers here is letting go of some of its restrictions: HEMP optimizes for one split and one frequency per performance domain, and statically keeps it like this for the whole duration of its runtime.

To the best of the author's knowledge, dynamic workload splitting in combination with DVFS has not previously been used to globally minimize energy costs in an online HMP setting with tunable performance constraints.

[10] It is in theory possible to prove that a reinforcement learning policy is optimal by first mathematically providing a lower bound for what energy cost *at least* needs to be, and showing that the algorithm matches that, but that is not done in any of the listed above studies.

4 Method

We will now take a look at the actual details that go into DREUS, how exactly it is modeled and how we will go about testing it against the actual current Linux scheduler.

4.1 Problem Formulation

DREUS aims to split utilization between P- and E-cores in such a way that average energy consumption is minimal, while still guaranteeing a certain performance, globally as well as locally. For each different amount of utilization that can run on a system, we provide an optimal split and frequency selection for each domain, alongside a realistic energy cost and performance associated with that split and frequency selection.
We first give a sketch of the optimization problem DREUS is designed to solve, afterwards going into more detail for each of the components, and what we need to model them specifically.

To rigorously note our optimization problem, we introduce a few mathematical definitions: let $\#E$ and $\#P$ respectively be the number of E- and P-cores of the CPU. We now let U_e, U_p and U respectively be the different amounts of util that can run on an E-core, P-core and the entire CPU. Similarly we also define F_e and F_p respectively as the set of all frequencies supported by E-cores, and all frequencies supported by P-cores. Now the task for DREUS becomes finding four functions: $s_e : U \mapsto U_e$, $s_p : U \mapsto U_p$ (with the property that for all $u \in U$ we have that $\#E \cdot s_e(u) + \#P \cdot s_p(u) = u$), $f_e : U \mapsto F_e$ and $f_p : U \mapsto F_p$, such that energy is minimal while respecting our performance constraints. Here if we have a certain util $u \in U$ that is running on the CPU, $s_e(u)$ dictates how much of that util should be moved to a single E-core, and likewise $s_p(u)$ dictates how much util should be running on a single P-core. Furthermore, f_e(u) dictates what frequency to run the E-cores on, and $f_p(u)$ the frequency for P-cores.

Note that this assumes that equally balancing util across a single performance domain is optimal. Another approach could be to balance util between just a subset of all cores in a single domain, and letting the others stay idle, which could theoretically save energy. However, even though cores in idle (sleep) states save significant energy, for cores to fall in even deeper sleep states we need the whole domain to be idle (group idle), which saves even more energy. As such it is generally assumed that it is most efficient to split util across a single domain, so that the entire domain can reach group idle as fast as possible (see the comment above `find_energy_efficient_cpu` in [45, `/kernel/sched/fair.c`]).

We first consider our condition of minimizing energy consumption. For this we define a map energy: $U_e \times U_p \times F_e \times F_p \mapsto \mathbb{R}$ that returns the energy cost associated with certain splits and frequencies. As we need to determine average energy over time, we also need a probability density function of U over time: we define $p_U : U \mapsto \mathbb{R}_+$ in such a way that for a certain subset $V \subseteq U$, we have that $\int_V p_U(u)\, du$ represents the percentage of time that the system spends with a reported

util that belongs to V.[11] The expression we want s_e, s_p, f_e and f_p to minimize now becomes:

$$\int_U p_U(u)\ \text{energy}(s_e(u), s_p(u), f_e(u), f_p(u))\ du.$$

We now consider performance constraints. For our purposes we measure performance in *util sojourn time*, which we define as the time a single unit of util has to wait before it is processed. We use this as our metric for performance, as something like makespan (total time it takes to process a complete workload) would simply not make sense: DREUS explicitly accounts for situations where cores are *not* completely utilized, which means cores are constantly waiting for tasks to unblock. So even if we would disregard energy in its entirety and fully focus on optimizing performance, total makespan would still stay roughly the same.

We now split sojourn time into two metrics: one for *average sojourn time*, and one for *worst-case sojourn time*. The former indicates the expected time a unit of util has to wait until it is processed across the whole CPU, and the latter indicates the time the last processed unit of util across the whole CPU has to wait.

We now discuss how we constrain DREUS with each metric: we firstly want DREUS to manage a constant global average sojourn time σ_μ: this way a unit of util at a random point in time will have to wait σ_μ units of time before it is processed. We define the map $\text{performance}_\mu : U_e \times U_p \times F_e \times F_p \mapsto \mathbb{R}$ that returns the average util sojourn time resulting from a certain split and frequency selection. Now we have that for s_e and s_p the following statement must hold:

$$\int_U p_U(u)\ \text{performance}_\mu(s_e(u), s_p(u), f_e(u), f_p(u))\ du\ \ \leq\ \ \sigma_\mu.$$

The second performance constraint comes in the form of a maximum $\sigma_{\max}$ a unit of util has to wait before it is processed at any point in time. We define $\text{performance}_{\text{wc}} : U_e \times U_p \times F_e \times F_p \mapsto \mathbb{R}$ such that it maps a certain split and frequency selection to its corresponding worst-case sojourn time. Note that if we want this constraint to be impactful, it is unavoidable that there are cases where it is impossible to keep the worst-case sojourn time under $\sigma_{\max}$, for example, when the CPU is fully utilized, there is only one way to split the util between the E- and P-cores (utilize both fully as well), which corresponds to the maximum possible worst-case sojourn time, which is certainly more than $\sigma_{\max}$[12]. If we let $\text{performance}_{\text{best}} : U \mapsto \mathbb{R}$ be the smallest worst-case performance that can theoretically be obtained for a certain util score $u \in U$, then we denote the set of possible util scores for which there *does exist* a split and frequency selection that satisfies this constraint as $U_{\text{feasible}} = \{u \in U \mid \text{performance}_{\text{best}}(u) \leq \sigma_{\max}\}$.

For all $u \notin U_{\text{feasible}}$, we simply require $\text{performance}_{\text{wc}}(s_e(u), s_p(u), f_e(u), f_p(u)) = \text{performance}_{\text{best}}(u)$.

[11] We need this probability density function, because if the system on average spends twice as long at a certain reported util u_1 than at another util u_2, we want the energy spent at u_1 to be weighted twice as heavy as the energy spent at u_2 when calculating an overall average.

[12] If this is not the case, this means that $\sigma_{\max}$ is greater than what the worst-case sojourn time can theoretically be, and this constraint becomes meaningless.

When we now put all these constraints together, the full optimization problem DREUS has to solve becomes:

$$\begin{aligned}
\text{minimize} \quad & \int_U p_U(u)\ \text{energy}(s_e(u), s_p(u), f_e(u), f_p(u))\ du, \\
\text{subject to} \quad & \int_U p_U(u)\ \text{performance}_\mu(s_e(u), s_p(u), f_e(u), f_p(u))\ du\ \leq\ \sigma_\mu, \\
\text{and} \quad & \forall u \in U_{\text{feasible}}, \quad \text{performance}_{\text{wc}}(s_e(u), s_p(u), f_e(u), f_p(u))\ \leq\ \sigma_{\max}, \\
\text{and} \quad & \forall u \in U \backslash U_{\text{feasible}}, \quad \text{performance}_{\text{wc}}(s_e(u), s_p(u), f_e(u), f_p(u)) = \text{performance}_{\text{best}}(u).
\end{aligned}$$

4.2 Linux-Specific Instantiation

Our model will be tested on an Intel Raptor Lake CPU (more on this in Section 4.5), as a replacement for EAS in Linux. The following section will concretely ground the above described problem into a Linux environment, by defining the actual optimization domain and the form of the involved functions.

4.2.1 Utilization Domain

We firstly specify the domain of different amounts of util that can run on the different parts of the system, namely U, U_e and U_p. As discussed in Section 2.3.3, there is already a good distinguished way of keeping track of util in the Linux kernel. As the Linux kernel only works with integer math, this very intuitively gives way to defining these domains: if CAP_E is the maximum capacity of an E-core, then we get $U_e = [0, \text{CAP\_E}]_{\mathbb{N}}, U_p = [0, 1024]_{\mathbb{N}}$ and $U = [0,\ \#E \cdot \text{CAP\_E} + \#P \cdot 1024]_{\mathbb{N}}$. So concretely, our Raptor Lake machine has 8 P-cores, and 12 E-cores with a maximum capacity of 623, and thus we have $U_e = [0, 623]_{\mathbb{N}}, U_p = [0, 1024]_{\mathbb{N}}$[13] and $U = [0, 15668]_{\mathbb{N}}$.

Because the number of possible util scores naturally gets discretized, the integral turns into a much simpler sum. In theory, the probability density function gets turned into a probability mass function, but for consistency, we leave the notation p_U as is. So in this concrete example, the equations for average energy consumption and average responsiveness become:

$$\sum_{u=0}^{15668} p_U(u)\ \text{energy}(s_e(u), s_p(u), f_e(u), f_p(u)) \quad \text{and}$$
$$\sum_{u=0}^{15668} p_U(u)\ \text{performance}_\mu(s_e(u), s_p(u), f_e(u), f_p(u)).$$

We also note that a Raptor Lake CPU supports frequencies ranging from 800 MHz to 4.2 GHz for E-cores, and from 800 MHz to 5.4 GHz for P-cores. Important here is that under the hood the Intel driver lets the Linux kernel choose between a discretized set of P-states, which is simply the target frequency divided by a constant. For E-cores, this constant is 100000, and for P-cores this is

[13] In actuality, 6 of the 8 P-cores have a maximum capacity of 1009 instead of 1024, but we ignore this for simplicity's sake.

78741 [45, `/drivers/cpufreq/intel_pstate.c`]. For our purposes, this translates to a minimum P-state of 8 for E-cores ($8 \cdot 100000 = 800000$) and 11 for P-cores ($11 \cdot 78741 = 866151$), and similarly a maximum P-state of 42 for E-cores, and 69 for P-cores. This leaves us with $F_e = [8, 42]_{\mathbb{N}}$ and $F_p = [11, 69]_{\mathbb{N}}$.

It is important to keep in mind that our utilization domain assumes that system utilization will always be perfectly split between the E- and P-cores, but as we only schedule tasks at wake-up, and because we only have a discrete number of tasks we can split between cores, in actual deployment most of the time the actual distribution of util among the cores will be slightly different from the optimal split. Similarly, as Intel processors allow each logical core to run at its own clock frequency, sometimes certain cores will run above their optimal P-state to avoid overutilization, even though our model assumes all cores in a performance domain run at the same frequency at all times.
As DREUS is designed to operate in scenarios with many tasks that block frequently (interactive workloads that do not overutilize the CPU), we accept this discrepancy for the sake of simplifying analysis.

4.2.2 Probability Mass Function

The next step in grounding DREUS into a real-world environment is deducing an appropriate probability function for p_U to take on. In an actual production environment these distributions would not need to be crafted manually: the kernel would keep track of reported util scores in a frequency distribution table over time[14] which we will call `time_in_util`, and use that as the input distribution for solving DREUS. We assume that the true workload over time is sufficiently static, such that if reported util is polled for long enough, the frequency table steadily converges towards the true probability distribution. We take into account changing circumstances[15] by making `time_in_util` an EWMA, just like PELT (but with a much longer half-time), and also storing the frequency distribution used for the last instance of DREUS, such that if `time_in_util` differs too much from this frequency distribution, DREUS solves itself again with this new distribution.
This is necessary, as DREUS can make inefficient decisions if it is solved on inaccurate data: energy consumption might not be minimal given the constraints, and the constraints may even be violated in the online environment.
In Section 4.3 we will briefly go over the effects on DREUS if solved on an inaccurate p_U, but for our experiments we will use a static load-independent p_U, based on the CPU/system environment that is tested on. We consider extensive research into what the above-discussed threshold should be, and DREUS's sensitivity with respect to an inaccurate frequency distribution as a topic for further research.

We must also note that keeping track of p_U is only effective because CPU utilization is invariant across different scheduler policies, provided the system does not overutilize. As tasks should in theory need the same amount of CPU time regardless of how fast they are processed, they should also always report the same util score at a certain point in time. The only time when this breaks is when a core overutilizes and the PELT score of a task stops being representative of its actual

[14] In the same way it also keeps track of used clock frequencies in a similar table, which is exposed to the user in `/sys/devices/system/cpu/cpu*/cpufreq/stats/time_in_state`.

[15] Say a user getting a new job and having to use new intensive software on their laptop regularly.

load. So as long as we try to minimize overutilization (which DREUS does), p_U should be relatively insensitive to a change in policy, and re-solving and deploying a new instance of DREUS will not change the accuracy of `time_in_util`.

4.2.3 Energy Consumption

We now take a look at the energy map. As discussed in Section 2.3.4, the current energy model provided by the Linux kernel is not accurate enough for DREUS to accurately optimize for actual energy consumption. This means we will have to build our own energy table.

For this we will use a built-in hardware interface called Intel RAPL to log energy readings. Using `stress-ng` we can consistently utilize each core to a specified percentage. We will cycle through a subset of all possible util scores to load evenly onto each performance domain, combined with a predetermined P-state for each domain with the use of the `userspace` governor. We will hold onto each combination for a few seconds, read its energy usage with RAPL, and let the CPU cool down again slightly for the next combination.
We assume an additive relation between P-core energy usage and E-core energy usage. So we get:

$$\text{energy}(u_e, u_p, f_e, f_p) \;=\; \text{static} \;+\; \text{dynamic}_e(u_e, f_e) \;+\; \text{dynamic}_p(u_p, f_p).$$

It is important to note that this assumption neglects factors for energy consumption that are shared across the entire CPU package, like uncore energy consumption. So the shape of our function will differ slightly from real life energy usage. However, to make the energy modeling feasible (in light of time constraints, among other things) we still adopt this design choice.

We start with determining static power usage by simply measuring power draw with RAPL while the system is idle. We will then gather all data points where util on the E-cores is zero, and similarly where util on P-cores is zero. This way we can isolate the effects of both dynamic_e and dynamic_p and determine their shapes. Lastly we gather some more data where E- and P-core utilization is intertwined, and use that to help fit the parameters and construct our final energy model.
Afterwards we broadly test our model by checking its R^2 value on some data we did not use for fitting.

Because we use Intel for our tests, we have a few important things to note:
First is the HWP used by Intel, described in Section 2.2. To combat this we set `intel_pstate` to passive, which hands control over the selected P-state to the kernel.
Second of all, Intel uses SMT to increase parallelism for their CPUs. As noted earlier, EAS also does not run on a machine with SMT enabled (Section 2.3.4), so we have to turn this off.
Lastly, we note that different instructions have different energy costs [46], which makes it hard to assign a single accurate energy cost to a specific core utilization. To solve this we use `stress-ng` with the `bitops` method, which makes sure the cores are stressed purely under bitwise operations, making our energy model more consistent. We just have to note that this means our energy model might undervalue possible uncore energy usage (e.g. energy used by RAM or cache), together with the fact that the experimental results will not be entirely reflective of actual CPU usage.

4.2.4 Performance

Lastly we try to model the different measures for performance we set up in Section 4.1. Luckily, because we measure performance as "util processed per unit of time", and the notions of util and capacity are deeply rooted in the Linux kernel, there is a very natural sense of going about this. As we already noted in Section 2.3.3, we can look at the capacity of a core as the "amount of util it can process in a time window". In our case, this time window would be 1024 μs. We can use this to properly define the different performance metrics we set up.

We start with performance$_\mu$. We defined this as the average time a single unit of util has to wait before it is fully processed.
What we mean by "a single unit of util" here, is an amount of load on a CPU that would cumulate exactly one util via PELT. So in practice, we define it as a quantity of load, equal to the amount of load a P-core can process at its maximum frequency in one μs. The unit of time we will use for our metric is one time window (so 1024 μs), meaning that if a unit of util has to wait one full time window on average, we would get performance$_\mu = 1$.

For our performance metrics we treat each core as a fluid server with its assigned util as its backlog, which drains fluid at its capacity. This is an idealization: in reality, the actual moment tasks need their CPU time is divided randomly throughout the time window, and task-reported util is always only a prediction of the average amount of CPU time it needs in a single window. In reality, most tasks utilize the CPU fully for a few time windows, before sitting idle for a few windows again. For the sake of simplifying analysis though, we assume that if a task reports a certain util score at the start of a time window, then that task will need the exact percentage of CPU time the util score would suggest, and it is requesting it from the start of the time window.

So in a concrete example, if a core has a capacity of 600, and a util of 300 (so it is loaded with 300 units of util), it would take half the time window to process all its util, and would sit idle for the other half of the window. Because by design each unit of util takes exactly the same amount of time to process, the average time it has to wait to be processed is just the time it takes for the core to process all its util, divided by 2. In our case that would thus be a quarter of a time window, resulting in a performance of 1/4.
We can calculate the capacity a core will have given a certain P-state. As mentioned, P-states scale linearly with frequency, which in turn means P-states also scale linearly with capacity. This gives us the ability to get the average performance for each domain. If we define the maximum P-state for respectively E-cores and P-cores as $f_{e,\max}$ and $f_{p,\max}$, we can calculate the average sojourn time of a unit of util to be:

$$\text{performance}_\mu(u_e, u_p, f_e, f_p) = \left(\#E \cdot u_e \cdot \frac{u_e \cdot f_{e,\max}}{2 \cdot f_e \cdot \text{CAP\_E}} + \#P \cdot u_p \cdot \frac{u_p \cdot f_{p,\max}}{2 \cdot f_p \cdot 1024} \right) / (\#E \cdot u_e + \#P \cdot u_p).$$

We quickly note that for $u_e = u_p = 0$ the denominator is zero. For this case we let performance$_\mu(0, 0, f_e, f_p) = 0$.
We now take a look at performance$_{\text{wc}}$. This is defined as the most time that a single unit of util has to wait before it is processed. This simply becomes:

$$\text{performance}_{\text{wc}}(u_e, u_p, f_e, f_p) = \max \left\{ \frac{u_e \cdot f_{e,\max}}{f_e \cdot \text{CAP\_E}}, \frac{u_p \cdot f_{p,\max}}{f_p \cdot 1024} \right\}.$$

Note that this is equivalent to the highest percentage of utilization that is running on any core. Given our Linux environment, this also gives us a very natural value to set $\sigma_{\max}$, namely 80%, as EAS gets taken over by EEVDF at that point anyway. We will thus copy this for our implementation: if a core gets overutilized, DREUS will disable itself, and EEVDF will take over.

Lastly, we consider $\text{performance}_{\text{best}}$, and the set U_{feasible} that gets created alongside it. As $U \backslash U_{\text{feasible}}$ is simply the set of CPU util scores where at least one core *must* be overutilized, we get that $U \backslash U_{\text{feasible}} = \{u \in U \ : \ u > \#P \cdot 0.80 \cdot 1024 + \#E \cdot 0.80 \cdot \text{CAP\_E}\}$, and from that it follows that $U_{\text{feasible}} = \{u \in U \ : \ u \leq \#P \cdot 0.80 \cdot 1024 + \#E \cdot 0.80 \cdot \text{CAP\_E}\}$.
Now we just need to calculate $\text{performance}_{\text{best}}$ for when $u \in U \backslash U_{\text{feasible}}$. When this is the case we automatically have $f_e = f_{e,\max}$ and $f_p = f_{p,\max}$. To determine $\text{performance}_{\text{best}}$, we need to minimize $\text{performance}_{\text{wc}}$ given a certain $u = \#E \cdot u_e + \#P \cdot u_p$, which gives:

$$\text{performance}_{\text{wc}}(u_e, u_p) = \text{performance}_{\text{wc}}\left(u_e, \frac{u - \#E \cdot u_e}{\#P}\right) = \max\left\{\frac{u_e}{\text{CAP\_E}}, \frac{u - \#E \cdot u_e}{1024 \cdot \#P}\right\}.$$

If we now deduce that for all $u_e \in U_e$ we have $\partial_{u_e} \frac{u_e}{\text{CAP\_E}} > 0$ and $\partial_{u_e} \frac{u - \#E \cdot u_e}{1024 \cdot \#P} < 0$, we can conclude that $\text{performance}_{\text{wc}}$ is at a minimum when $\frac{u_e}{\text{CAP\_E}} = \frac{u - \#E \cdot u_e}{1024 \cdot \#P}$ (since any change from that equilibrium lets either the LHS or RHS grow). So we get:

$$u_e\left(\frac{1}{\text{CAP\_E}} + \frac{\#E}{1024\#P}\right) = \frac{u}{1024\#P} \implies u_e = \frac{u \cdot \text{CAP\_E}}{1024\#P + \text{CAP\_E}\#E}.$$

Which then gives us:

$$\text{performance}_{\text{best}}(u) = \frac{u}{1024\#P + \text{CAP\_E}\#E}.$$

We now finish this section with a quick recap on how DREUS will concretely look in this Linux implementation (where $u_{\max} = 1024\#P + \text{CAP\_E} \cdot \#E$):

$$\begin{aligned}
\text{minimize} \quad & \sum_{u=0}^{u_{\max}} p_U(u)\ \text{energy}(s_e(u), s_p(u), f_e(u), f_p(u)), \\
\text{subject to} \quad & \sum_{u=0}^{u_{\max}} p_U(u)\ \text{performance}_{\mu}(s_e(u), s_p(u), f_e(u), f_p(u)) \ \leq \ \sigma_{\mu}, \\
\text{and} \quad & \forall u \in U_{\text{feasible}}, \quad \text{performance}_{\text{wc}}(s_e(u), s_p(u), f_e(u), f_p(u)) \ \leq \ 0.8, \\
\text{and} \quad & \forall u \in U \backslash U_{\text{feasible}}, \quad \text{performance}_{\text{wc}}(s_e(u), s_p(u), f_e(u), f_p(u)) = \frac{u}{u_{\max}}.
\end{aligned}$$

4.3 Solving the Instantiation

To talk more easily about possible solutions, from now on we let X be our solution space, total_energy the energy function we try to minimize, and $\text{total\_performance}_{\mu}$ the performance constraint function

we must keep below σ_μ. So we have:

$$C(u) = \{(u_e, u_p, f_e, f_p) \in (U_e \times U_p \times F_e \times F_P) \mid 0 \leq \#E \cdot u_e + \#P \cdot u_p - u < \min(\#E, \#P) \\ \wedge \text{performance}_{\text{wc}}(u_e, u_p, f_e, f_p) \leq \max\{0.8,\ \text{performance}_{\text{best}}(u)\}\},$$

$$X = \{x \in \text{Map}(U, U_e \times U_p \times F_e \times F_P) \mid \forall u \in U,\ x(u) \in C(u)\},$$

$$\text{total\_energy}(x) \ = \ \sum_{u=0}^{u_{\max}} p_U(u) \cdot \text{energy}(x(u)),$$

$$\text{total\_performance}_\mu(x) \ = \ \sum_{u=0}^{u_{\max}} p_U(u) \cdot \text{performance}_\mu(x(u)).$$

Note that in our definition for $C(u)$, instead of requiring $\#E \cdot u_e + \#P \cdot u_p = u$ like in Section 4.1, we require $0 \leq \#E \cdot u_e + \#P \cdot u_p - u < \min(\#E, \#P)$. This is because in our real-life instantiation of DREUS, u, u_e and u_p are all modeled as integers, and as such, there are some $u \in U$ for which no $u_e \in U_e$ and $u_p \in U_P$ exist such that $\#E \cdot u_e + \#P \cdot u_p = u$. If without loss of generality, we let $\#E > \#P$, then essentially we deal with this by allowing a single split for each u_e, namely, the one that lets $\#E \cdot u_e + \#P \cdot u_p$ overshoot u the least (provided the choice for both u_e and u_p does not overutilize their respective cores).

Now that we actually have a concrete model to solve, we will solve it for various σ_μ using a technique called *Lagrangian relaxation*. Here, instead of minimizing total_energy with $\text{total\_performance}_\mu \leq \sigma_\mu$, we choose a certain $\lambda > 0$ and minimize $\text{total\_energy} + \lambda \cdot \text{total\_performance}_\mu$.
So concretely we are minimizing total energy usage, with a certain penalty for total_performance. This is useful because of the following: say we have an instance $x_\lambda \in X$ that solves this relaxation. We now take any other $x \in X$ with $\text{total\_performance}_\mu(x) \leq \text{total\_performance}_\mu(x_\lambda)$. So this then gives:

$$\lambda \cdot \text{total\_performance}_\mu(x) \leq \lambda \cdot \text{total\_performance}_\mu(x_\lambda).$$

But as x_λ is the solution to the relaxation, by definition we must also have:

$$\text{total\_energy}(x) + \lambda \cdot \text{total\_performance}_\mu(x) \geq \text{total\_energy}(x_\lambda) + \lambda \cdot \text{total\_performance}_\mu(x_\lambda).$$

Which must mean that:

$$\text{total\_energy}(x) \geq \text{total\_energy}(x_\lambda).$$

We conclude that x_λ is also the solution to the original problem with performance constraint $\text{total\_performance}_\mu(x_\lambda)$. If we now notice the fact that $\text{total\_performance}_\mu(x_\lambda)$ is non-increasing as a function of λ (if we induce a higher penalty on performance, the util sojourn time will definitely not turn out higher), we can simply sweep across increasing values of λ and solve the relaxation for each. This then gives us a family of operating points that are Pareto-optimal with respect to performance and our energy model. To find a specific value of σ_μ, we can apply binary search when solving the Lagrangian relaxations.

To solve an instance of our Lagrangian relaxation we note that for every $x \in X$:

$$\begin{aligned}
&\text{total_energy}(x) + \lambda \cdot \text{total_performance}_\mu(x) \\
&= \sum_{u=0}^{u_{\max}} p_U(u) \cdot \text{energy}(x(u)) + \lambda \cdot p_U(u)\ \text{performance}_\mu(x(u)) \\
&= \sum_{u=0}^{u_{\max}} p_U(u) \cdot \left[\text{energy}(x(u)) + \lambda \cdot\ \text{performance}_\mu(x(u))\right].
\end{aligned}$$

We conclude from this that if for every u, $\text{energy}(x(u)) + \lambda\ \text{performance}_\mu(x(u))$ is minimal, then $\text{total\_energy}(x) + \lambda \cdot \text{total\_performance}_\mu(x)$ must also be minimal.
This means that to solve the relaxation, we simply go through all $u \in U$ one by one, and calculate which configuration $c_u \in C(u)$ gives the minimal $\text{energy}(c_u) + \lambda\ \text{performance}_\mu(c_u)$ by brute force, and set $x_\lambda(u) = c_u$. This leaves us with a map $x_\lambda \in X$ that solves our relaxation.

Note that our solution x_λ is completely independent of our probability mass function p_U. The only influence p_U has on DREUS is how x_λ is mapped onto $\text{total\_performance}_\mu(x_\lambda)$, so if DREUS is being solved on a p_U that differs from the true PMF, the resulting solution will still be energy-minimal for the actual performance the solution runs on. The actual performance will just not match up with the desired σ_μ.

After DREUS is solved, a lookup table is created that returns the optimal split at a certain util u. With this lookup table, we can finally actually inject DREUS into the Linux kernel.

4.4 Implementing DREUS

To actually run DREUS in a Linux environment, we implemented DREUS by replacing the `find_energy_efficient_cpu` call in `select_task_rq_fair` with a static call to either `find_energy_efficient_cpu`, or our own function `dreus_dispatcher`. The function the static call points to can be changed with a kernel module. This way DREUS can be turned on and off without having to recompile the whole kernel. In general, we expose the following API that can be used in kernel modules:

- `int set_eas_dispatcher(int func)`:
  Turn DREUS on or off the kernel. If `func=0`, set `find_energy_efficient_cpu` as the dispatcher. If `func=1`, use `dreus_dispatcher`. Returns 0 on success.

- `int set_dreus(void* table, size_t nr_entries, int prev_margin, bool enable_gov)`:
  Change the table DREUS queries from, and also allow DREUS to take over `schedutil` in choosing the P-states via `enable_gov`. `prev_margin` sets the margin for which DREUS favors cache locality by scheduling a task on the core it previously ran on. It is not used in this thesis, but is still implemented for possible further research. Returns 0 on success.

- `int disable_dreus()`:
  Remove the DREUS table from kernel memory, and give P-state control back to `schedutil`. Returns 0 on success.

Now, if DREUS has been loaded in, a table has been set and DREUS has taken over `schedutil`, DREUS is active in two places: the dispatcher is active inside `kernel/sched/fair.c`, via `dreus_dispatcher`, and the governor is active inside `kernel/sched/cpufreq_schedutil.c`, adding a few lines of code to the already existing function `sugov_update_single_perf`.

Inside `dreus_dispatcher`, the flow is as follows: when a task t wakes up, we obtain the total util across the whole CPU by cycling through all cores to find the core with the lowest util of each performance domain, while simultaneously summing all util scores.
We then query the lookup table described in Section 4.3 to find the optimal split for that summed util score. After obtaining this split we check which of the lowest util cores is farthest away from that optimal split (we look for the core c that maximizes `optimal_util`-Util_c), and place t at that core.

Inside `sugov_update_single_perf`, schedutil normally calls the architecture-specific driver with a minimum desired capacity for that core (so 1.25 times the util), which the driver then translates to an actual frequency, or (in our case) P-state. If the DREUS governor is enabled however, the lookup table is once again queried based on the util of the entire system at that moment, and the resulting P-state is translated to the capacity that core would obtain for that P-state. We then choose the highest of the capacities chosen by schedutil and DREUS to call the driver. We choose the highest of the two here, since we can have the situation where a core is more utilized than its optimal split. So if necessary, we fall back to the capacity chosen by schedutil to avoid the core becoming overutilized.

4.5 Experiment Setup

We test DREUS by compiling the Linux kernel 7.2.0 (Ubuntu 26.04) with our patch, on a machine running on an OptiPlex Tower Plus 7020 with an Intel i7-14700 CPU.

An important consideration when choosing our benchmark is that we cannot use any benchmarks that overutilize the CPU often, as it is critical that the benchmark provides a utilization distribution that comes relatively close to real end-user computer/mobile usage. On top of that, if one of the cores gets overutilized, DREUS (just like EAS) turns off and EEVDF takes over.

As such for our benchmark we will be replaying the publicly available NASA web traffic logs from 1995, first analyzed in [6]. The entries in this log are bursty in arrival times, and the amount of traffic per second has great variance over time [5]. These properties can also be seen in end-user computing [22].
We simulate the replay with two Python scripts: `replay.py` and `server.py`. `replay.py` parses all logs and sends the entries to `server.py` via HTTP, with the number of bytes sent in the reply by NASA as the payload. `replay.py` includes a `speedup` argument that lets the user replay the trace a certain factor faster/slower.
`server.py` then treats each connection (all HTTP requests from a unique IP) as a separate task, and for each HTTP request, does some amount of work based on the number of bytes in the payload, and sends a response back to `replay.py`, which in return logs the time between the request and the response. Because we run both `server.py` and `replay.py` on the same machine, in theory

the average sojourn time of a request given the logs from `replay.py` is an equivalent performance metric to performance$_\mu$ (both are multiples of sec/instr). This does not translate perfectly to real life, because of factors like memory-boundedness, but pragmatically, we consider the average sojourn time to reflect our choice of σ_μ.
Running both scripts on the test machine also means that `server.py` will contribute to the total CPU load, but as this is symmetrical across configurations this does not induce a bias in the comparisons between them.

This setup gives us multiple factors to scale the workload and number of tasks on: we can speed up the replay, and increase the amount of work done per task. For our experiments, we replay the trace at 600 times the original speed, and for each incoming request with a payload of B, the server encrypts a key through $1000\sqrt{B}$ iterations of HMAC-SHA256. We chose these factors empirically, such that substantial variance in total utilization over time is provided, while the entire system is not overutilized too often.
NASA provides two log files: one with logged HTTP requests from July 1 to July 31, and a second one with logged requests from August 4 to August 31. We will replay the logs from July and trace them using a tracing system developed by Google, under the name Perfetto. We then reverse-engineer total system utilization after every 1024 μs and keep track of how many times each utilization occurs, thereby building a frequency table which we will use for our PMF p_U. We then split the August logs into 12 parts, each with the same number of entries, and use the first 5 to test DREUS.

Before we can test DREUS we first solve it for various σ_μ via the method described in Section 4.3. We solve for four instances of DREUS in total, namely the following:

- DREUS_NRG: We disregard the performance constraint (or equivalently, set $\sigma_\mu = 0.5$), and optimize purely for energy minimization.
- DREUS_25: We solve DREUS with $\sigma_\mu = 0.25$.
- DREUS_20: We solve DREUS with $\sigma_\mu = 0.20$.
- DREUS_15: We solve DREUS with $\sigma_\mu = 0.15$.

We then test each DREUS instantiation by loading the respective table into the kernel, running every instantiation with each of the 5 split NASA logs, calculating energy drainage by polling Intel RAPL's CPU package energy counter before and after each test, and calculating the average response time for each request. We also test each NASA log with EAS by offloading DREUS.

The goal of this experiment will be to test how different configurations of DREUS hold up against EAS, and whether energy consumption rises monotonically as σ_μ lowers, and similarly, whether sojourn time lowers monotonically as σ_μ lowers.
We hypothesize that all DREUS configurations consume more energy and have a lower average sojourn time when compared to EAS, as the DREUS governor never picks a lower P-state than schedutil. Furthermore we indeed expect energy usage to rise and sojourn time to fall as we test for DREUS configurations with a lower σ_μ.

For statistical rigor, we run each test eight times and calculate the average and standard deviation for each result. After each test, we let the CPU cool back down and erase the cache using `/proc/sys/vm/drop_caches`, so that test results do not get muddled by earlier runs.

5 Experiments

In this section we will look at the results from our experiments, first examining the probability mass function we attained from the first month of NASA logs, then looking at the finalized energy model, and then finally analyzing the results of the various DREUS instantiations against EAS.

5.1 Probability Mass Function

After tracing the replay of the NASA logs from July, and rebuilding total system utilization after each 1024 μs time window, we can see the end result in the histogram in Figure 2. We can

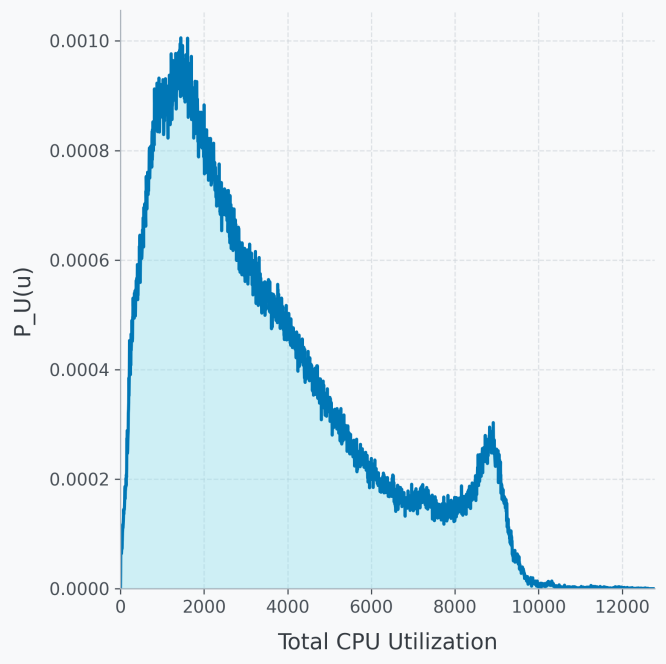
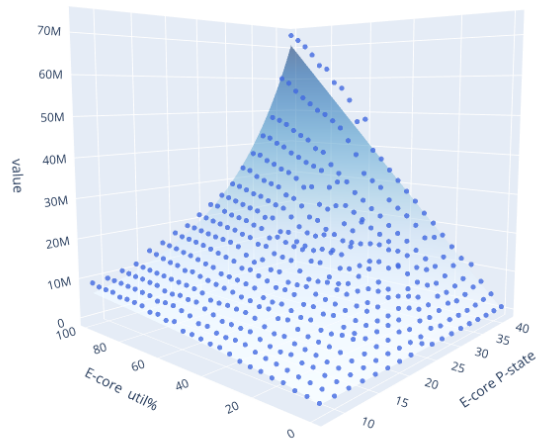


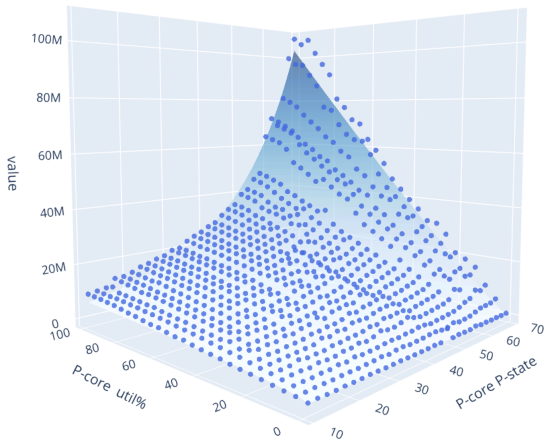
Figure 2: The PMF p_U for DREUS.

see significant variance in CPU utilization over time, peaking at around 2000 util, then steadily declining, up until 9000 util, where we have a new smaller peak. Higher utilization than 9000 rarely occurs, and CPU utilizations above 12622 were not observed at all. One could hypothesize that this hard cut-off indicates some sort of bottleneck on the original NASA servers, but as we only care about obtaining a reasonable PMF for DREUS, we will not discuss the cause of the function's form here.

Similarly, we will also not try to fit any underlying distribution over these results, and will simply plug these raw obtained values into DREUS. This is prone to overfitting, but as discussed in Section 4.3, this only leads to a minor shift away from the target σ_μ, and preserves energy minimality for the realized performance.



(a) Plot of E-core util% and P-state vs Energy in μW with fit. ($R^2 = 0.9847,\ n = 468$).



(b) Plot of P-core util% and P-state vs Energy in μW with fit ($R^2 = 0.9783,\ n = 780$).

Figure 3: Plots of C_0+dynamic$_e(u_e, f_e)$ and C_0+dynamic$_p(u_p, f_p)$.

Lastly, we also consider the fact that while tracing, a core was overutilized only 10% of the time. Although this number is dependent on the active dispatcher (EAS in this case), it does support the notion that when replaying the NASA logs, cores can avoid being overutilized if the dispatcher tries to avoid it.

5.2 Energy Model

For our energy model we first measure the static power draw of our machine. After 23 hours of measuring idle energy consumption we arrived at a baseline power draw of 325337 μW. We will refer to this baseline as C_0.

We follow this up by sweeping across a subset of all possible combinations of P-core utilization and P-core P-states, while letting the E-cores idle, and vice versa. The results are shown in Figure 3. If we take a 1D slice of one of these graphs where util% is set as a constant, we get a graph of the form $K \cdot \exp(r \cdot \text{Pstate})$. Similarly, when we take a 1D slice where Pstate is set as a constant, we see a regression of the form $K \cdot \text{util}\%^n$, where n approaches 1 as Pstate grows. This gives us the following ansatz for dynamic$_e(u_e, f_e)$ (and similarly for dynamic$_p$):

$$\text{dynamic}_e(u_e, f_e) = k_e \cdot \exp(r_e \cdot f_e) \cdot (\text{util}_e\%)^{a_e f_e + b_e}.$$

Here we have that $\text{util}_e\% = (u_e \cdot f_{\max,e})/(f_e \cdot \text{CAP\_E})$.

Using this our final energy model becomes:

$$\text{energy}(u_e, u_p, f_e, f_p) = C_0 + k_e \cdot \exp(r_e \cdot f_e) \cdot \left(\frac{f_{\max,e} \cdot u_e}{\text{CAP\_E} \cdot f_e}\right)^{a_e f_e + b_e} + k_p \cdot \exp(r_p \cdot f_p) \cdot \left(\frac{f_{\max,p} \cdot u_p}{1024 \cdot f_p}\right)^{a_p f_p + b_p}.$$

Training Data	n	R^2
E-cores	468	0.9847
P-cores	780	0.9783
P-states	1080	0.9753
util%	1320	0.9789
Overall	3648	0.9803

Table 1: Resulting R^2 value for different types of training data

We now fit k_e, r_e, a_e, b_e and k_p, r_p, a_p, b_p using the data from Figure 3, and 2400 more data points where u_e, f_e and u_p, f_p are mixed. For 1080 of these points we repeatedly set the util% of both cores, and sweep across the possible P-states, and for the other 1320 we set the P-state and sweep across the possible values for util%. This gave the following results:

$$\begin{aligned} k_e &= 244208, & r_e &= 0.0239791, & a_e &= 0.0105755, & b_e &= 0.566418, \\ k_p &= 479681, & r_p &= 0.00460011, & a_p &= 0.00982937, & b_p &= 0.418628. \end{aligned}$$

In Table 1 the coefficient of determination for our model is shown for each type of training data. Afterwards we tested our energy model on 3456 more data points we held out from training. This resulted in a coefficient of determination of $R^2 = 0.9455$. We conclude that our energy model sufficiently approximates energy in most cases, even though it does introduce a gap between the training and the test data.

We can explain this gap by taking note of one place where our model breaks down:
If we look at the partial derivative of (without loss of generality) f_e with respect to dynamic_e we get the following:

$$\begin{aligned} \frac{\partial \text{dynamic}_e(u_e, f_e)}{\partial f_e} &= k_e \cdot r_e \cdot \exp(r_e \cdot f_e) \cdot \left(\frac{f_{\max,e} \cdot u_e}{\text{CAP\_E} \cdot f_e}\right)^{a_e f_e + b_e} + \\ &\qquad k_e \cdot \exp(r_e \cdot f_e) \left(a_e \cdot \ln\left(\frac{f_{\max,e} \cdot u_e}{\text{CAP\_E} \cdot f_e}\right) - \frac{a_e f_e + b_e}{f_e}\right) \left(\frac{f_{\max,e} \cdot u_e}{\text{CAP\_E} \cdot f_e}\right)^{a_e f_e + b_e} \\ &= k_e \cdot \exp(r_e \cdot f_e) \cdot \left(\frac{f_{\max,e} \cdot u_e}{\text{CAP\_E} \cdot f_e}\right)^{a_e f_e + b_e} \left(r_e - a_e - \frac{b_e}{f_e} + a_e \cdot \ln\left(\frac{f_{\max,e} \cdot u_e}{\text{CAP\_E} \cdot f_e}\right)\right). \end{aligned}$$

Because $r_e - a_e - \frac{b_e}{f_e} < 0$ for our parameter selection and $f_e \in F_e$, if we want this partial derivative to be non-negative we must have:

$$\ln\left(\frac{f_{\max,e} \cdot u_e}{\text{CAP\_E} \cdot f_e}\right) \geq 0 \implies f_{\max,e} \cdot u_e \geq \text{CAP\_E} \cdot f_e.$$

This means that if $u_e \leq \frac{\text{CAP\_E} \cdot f_{\min,e}}{f_{\max,e}}$, we have that $\frac{\partial \text{dynamic}_e(u_e, f_e)}{\partial f_e} \geq 0$ for all $f_e \in F_e$. This would mean in our case that if $u_e \leq 118$, the most energy-efficient P-state according to our model is $f_{\max,e}$. A similar argument also follows for P-cores when $u_p \leq 163$. This can be justified by noting that a higher P-state means the cores can be idle longer, which could in theory lead to energy savings.

However, we discard these results for two reasons:

Firstly, we can see our energy model falls apart for these low util values: when setting P-states for both domains to the lowest available, and sweeping across util%, we have that our energy model returns $R^2 = -1.5383$. So our energy model is unreliable here.
Secondly, if we were to accept these results, DREUS would always pick the maximum P-state for a domain when we have $u_e \leq 118$ or $u_p \leq 163$. This can lead to undesirable rapid oscillation in clock speed when the optimal splits come close to these values.

Because of this, whenever we have $u_e \leq 118$, we let DREUS automatically pick the lowest available P-state that does not overutilize the E-cores. The same goes for when $u_p \leq 163$.

5.3 DREUS

5.3.1 Solver

Our solver is able to take in a $\lambda_{\min}$, a $\lambda_{\max}$ and a step size h, and will then calculate the solutions to the Lagrangian relaxation from Section 4.3 for all $\lambda_{\min} + k \cdot h$ with $k \in \mathbb{N}$ such that $\lambda_{\min} + k \cdot h \leq \lambda_{\max}$. Our implementation is built in C++, and is single-threaded. The author recognizes the solver could very well be sped up by implementing it in a multi-threaded fashion, but as can be seen below, the single-threaded implementation also suffices to solve DREUS in reasonable time. If n represents the different amounts of target relaxations our solver has to solve, then Table 2 shows the time and memory usage while solving DREUS for different values of n. The results of the table were collected by running our implementation on an i7-9750H processor with 32 GB of RAM.

n	time (s)	memory usage (KB)
1	112.9 ± 1.7	36208 ± 101
10	186.3 ± 14.4	37431 ± 141
100	547.0 ± 6.0	48384 ± 145

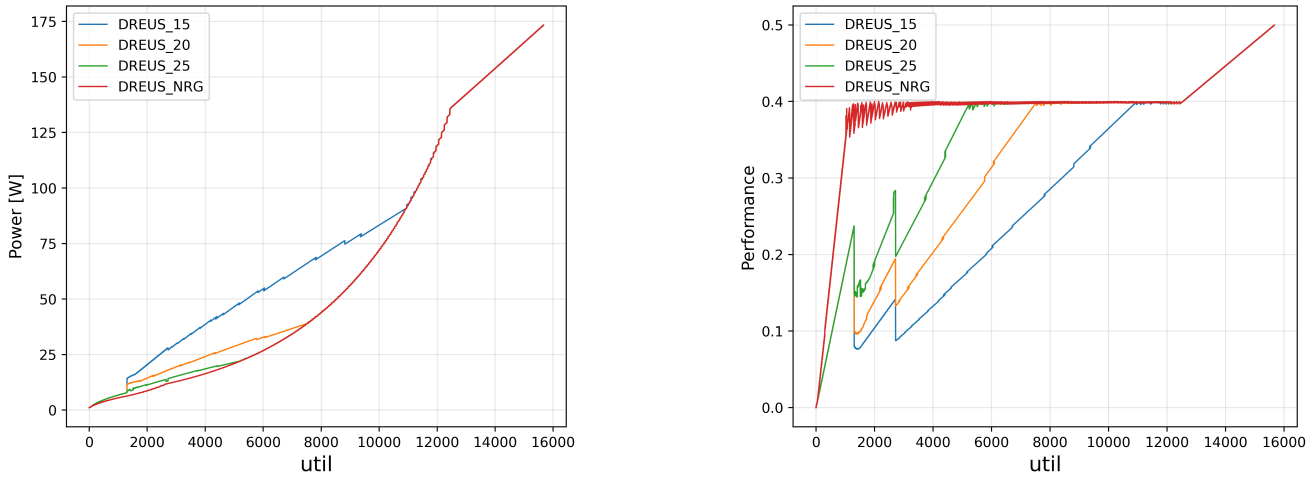
Table 2: Time and memory usage of the DREUS solver depending on the number of relaxations solved, $\pm$ shows 95% confidence intervals on the mean based on the t-distribution, n=8 runs.

We vastly reduced memory usage by only storing the energy and performance values for one $C(u)$ at a time, and immediately solving for the optimal $c \in C(u)$ that minimizes $\text{energy}(c) + \lambda \cdot \text{performance}_\mu(c)$ for all different values of λ we try to solve the relaxation for.

Table 3 shows the energy and performance of the different DREUS configurations, along with the λ that produced them.
We note that DREUS_25 does not have an exact performance of 0.25, but sits just $2 \cdot 10^{-6}$ below that. This happens because even though $\text{total\_performance}_\mu(x_\lambda)$ is non-increasing with respect to λ, our initial problem is discrete, and therefore a 1D sweep across different values of λ never retrieves all possible performances.
Because of this, we acknowledge that DREUS_25 may not actually be the most energy-efficient with $\text{total\_performance}_\mu \leq 0.25$, but our results do provide an upper bound for the energy consumption of this most energy-efficient model. Similarly, for $\lambda = 16.605$, we get an energy consumption

configuration	λ	avg. power (W)	performance
DREUS_NRG	0	16.0715	0.365723
DREUS_25	16.6051	17.5943	0.249998
DREUS_20	51.554	20.6387	0.200000
DREUS_15	197.038	30.8692	0.150000

Table 3: Solution metrics of different DREUS configurations, avg. power corresponds to total_energy(x_λ), performance corresponds to total_performance$_\mu(x)$.



(a) Energy per solution state.

(b) Performance per solution state.

Figure 4: Performance and energy costs at all CPU states for different DREUS configurations.

of 17.5942 W and a performance of 0.250003, which provides a lower bound. We conclude that DREUS_25 is theoretically at most 0.0005% of energy costs removed from the minimum with total_performance$_\mu \leq 0.25$.

In Figure 4 we can see how much energy and performance each DREUS solution reports at different CPU states. The first thing to notice in these plots is in Figure 4b, where at utils of 1305 and 2721 there is a big drop for all configurations except DREUS_NRG. This is because 1305 is the first util score where P-state choice is not strictly determined by the cut-off discussed in Section 5.2 ($8 \cdot 163 = 1304$). So from exactly this point on, more aggressive schedulers can immediately turn up the P-state for the P-cores. Similarly, util 2721 is the first where this is the case for both domains at the same time ($8 \cdot 163 + 118 \cdot 12 = 2720$). We can also see a slight ramp up in energy at util score 1305 in Figure 4a. Furthermore we can see in Figure 4b that performance grows linearly with util, until the threshold of 0.8 is reached. In Figure 4a, configurations besides DREUS_NRG also seem to let energy cost grow linearly with util starting from the ramp at 1304, until they have caught up with DREUS_NRG again (their performance has hit 0.4).

This research will not go into the reason behind this linearity, but we do take note of it for possible further research, as it may lead the way for more sophisticated solvers to use this property in streamlining the solving of DREUS configurations.

5.3.2 Experiment Results

We now finally take a look at the results of our main experiment, described in Section 4.5. After 8 iterations of each combination of DREUS/EAS with one of the 5 NASA logs, the results are combined by adding the average energy consumptions on each log, and taking the average of all average sojourn times on each log (we can do this, as each log contains the same number of requests). We also calculate the energy consumption that was expected by DREUS beforehand, based on the experiment duration, times predicted average power. The results can be seen in Table 4.

Configuration	Energy (kJ)	Expected Energy (kJ)	Sojourn time (ms)
EAS	104.62 ± 0.13	-	249.09 ± 0.33
DREUS_NRG	110.67 ± 0.09	33.55	235.95 ± 0.37
DREUS_25	110.44 ± 0.11	36.74	234.23 ± 0.34
DREUS_20	111.11 ± 0.09	43.09	226.57 ± 0.46
DREUS_15	119.21 ± 0.10	64.45	197.38 ± 0.23

Table 4: Aggregate results over all five NASA logs, $\pm$ shows 95% confidence intervals on the mean (T, $n = 8$)

We can see that while all configurations consistently perform worse on energy savings than EAS, they do outperform EAS with respect to average sojourn time. In general, for all our configurations, a lower σ_μ corresponds to a lower sojourn time. The differences are statistically significant, as none of the confidence intervals overlap with one another. In the same way we note that a lower σ_μ also coincides with a higher energy usage, with the exception of DREUS_NRG, which has a higher energy cost than DREUS_25, contrary to expectation. This way DREUS_NRG is strictly dominated by DREUS_25, which performs better with respect to energy *and* performance.
If we also take a look at the expected energy scores, we can see our model severely underestimates energy costs, with configurations with a loose performance constraint suffering the most from this (DREUS_NRG has a 229% mismatch, and DREUS_15 an 85% mismatch). This can be explained by the same reasons that were given in Section 4.2.3: the assumption of energy consumption between E- and P-cores being additive is not always accurate, and the energy model was trained on energy consumed by bitwise operations, even though the actual load that was used for testing had a very different instruction mix.

We also note, like we hypothesized, that every single DREUS configuration consumes more energy than EAS, and all improve on sojourn time with respect to EAS. This, however is enforced by the design: as per Section 4.4, DREUS always picks a P-state that is *at least* the P-state that would be picked by schedutil. As such DREUS is inevitably operating on the more performant and energy-draining side of EAS. Figure 5 shows the percentage difference between each configuration and EAS for each log, and for the combined total.

We did not keep track of overutilization frequency as part of our experiment. However, as DREUS never operates on lower P-states than schedutil and each configuration is designed to steer away from overutilization, there is reason to believe that DREUS does not overutilize cores substantially more than EAS. This, however, remains speculative, and real evidence for this fact is left for future research.

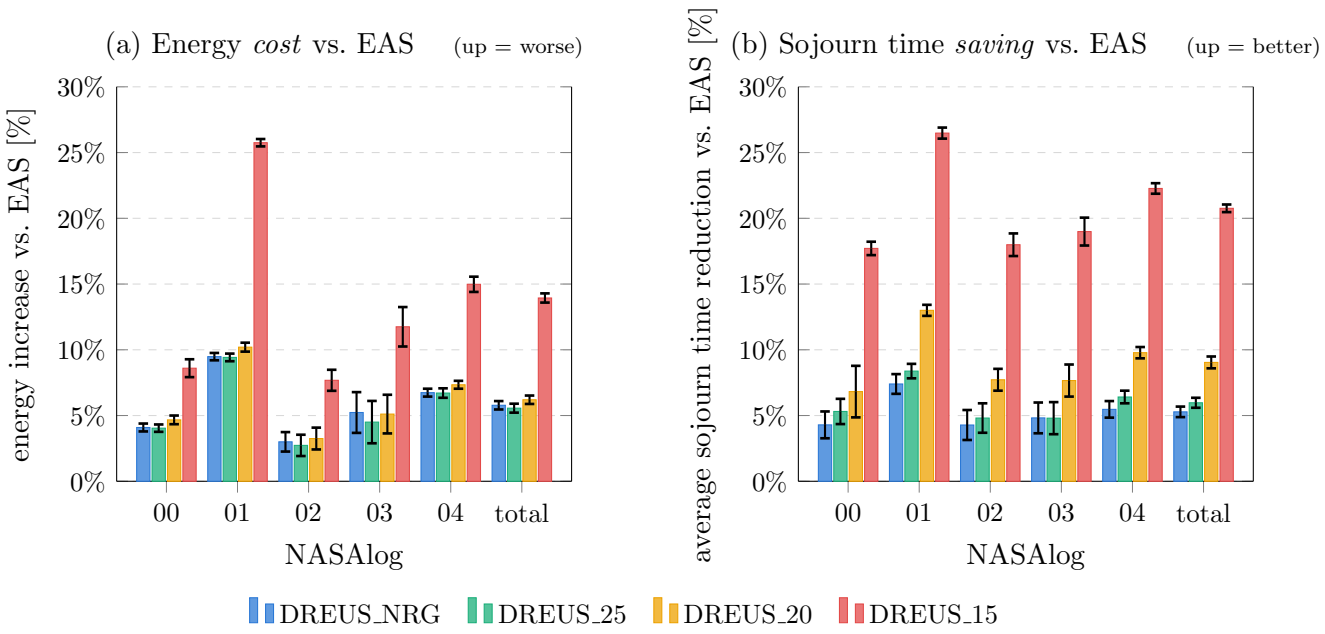


Figure 5: DREUS vs EAS, error bars show 95% confidence intervals (Welch, n=8).

6 Conclusions and Further Research

This thesis determined how different configurations of DREUS, a non-greedy dispatch policy with a tunable performance-energy trade-off, compare to EAS with respect to energy and performance. The experiments showed DREUS is able to provide a selectable operating point with respect to energy usage and performance, with lowering σ_μ leading to a statistically significant monotonic reduction of task sojourn time for the provided configurations. Compared to EAS, DREUS_15 decreased average sojourn time by 20.8%, while consuming 13.9% more energy. Similarly, DREUS_20 decreased sojourn time by 9.0% and increased energy costs by 6.2%. DREUS_25 decreased sojourn time by 6.0% and consumed 5.6% more energy. DREUS_NRG is strictly dominated by DREUS_25. We note that DREUS's range of operating points lies almost strictly on the more performant side of EAS, as DREUS's frequency governor always chooses *at least* the frequency that would have been chosen by schedutil. We conclude that under this design no configuration of DREUS was able to beat EAS on energy consumption.

The most important shortcoming of our experiment is the discrepancy between the energy model used to precompute the different DREUS lookup tables, and the actual energy costs at certain states: the model underestimated actual energy consumption by a factor of up to 3. The important thing to notice here is that this underestimation is much more pronounced for DREUS configurations with a higher value for σ_μ. The cause of this has been discussed in Section 5.2, where it was shown that for low P-states (more commonly chosen when σ_μ is high), the energy model becomes very unreliable, leading DREUS to potentially pick the wrong P-states/util splits. This can be very clearly inferred from the results: DREUS_NRG, designed to be the most energy-efficient possible configuration of DREUS, consumes more energy than DREUS_25. We can conclude that because of this, the operating points are not all Pareto-optimal in practice.

We also note that our choice of σ_μ does not concretely translate to an actual measurable cap on task sojourn time. Across different DREUS configurations, reducing σ_μ from 0.25 to 0.15 (40%) only resulted in a 16% reduction in measured sojourn time, so the two metrics are not actually equivalent. In the context of DREUS, however, it is more important that σ_μ acts as a scalar to monotonically improve performance, which it achieves.

Lastly, given DREUS's main purpose as an implementation of a scalable energy-performance trade-off in the Linux kernel, the experiment would have benefited from DREUS being tested against a greater sample of already available baselines, like the performance frequency governor or the userspace governor set at different frequencies. This would more concretely place the operating points our different DREUS configurations run at against those already available in the kernel. However, because of time constraints and machine unavailability, these experiments could unfortunately not be run.

Further research could explore developing more efficient ways of solving DREUS given the results in Section 5.3.1, more extensively researching DREUS's effectiveness under a mismatched workload distribution for p_U, implementing the re-solving mechanism described in Section 4.2.2, or using the methods introduced by DREUS for other optimization goals, like thermal reduction or hardware longevity.
Furthermore, another focus may lie in producing a more workable and accurate energy model for DREUS, along with researching whether energy consumption of DREUS can be brought down further with the already-implemented options of disabling the DREUS governor for high σ_μ, and accounting for cache locality with `prev_margin`.

We also take note of the fact that it was surprisingly hard to find a representative benchmark to simulate realistic interactive CPU usage for PC/desktop environments. Many benchmarks are meant to fully utilize the CPU, or do not vary in utilization. This leaves a gap in research for developing such a benchmark.

Taken together, the limitations most immediately concern the accuracy of the energy model and the number of comparisons to other algorithms, and do not affect our central findings. Therefore we conclude that DREUS offers a theoretically grounded algorithm for a nuanced way to balance performance with energy, which does transfer to an actual online environment: an adjustment in this balance can be seen in the resulting performance and energy measurements.

References

[1] Jazeem Abdul Jaleel, Sherwin Doroudi, and Kristen Gardner. Queue-length-aware dispatching in large-scale heterogeneous systems. *Queueing Systems*, 108(1):125–184, 2024.

[2] Jazeem Abdul Jaleel, Sherwin Doroudi, Kristen Gardner, and Alexander Wickeham. A general "power-of-d" dispatching framework for heterogeneous systems. *Queueing Systems*, 102(3):431–480, 2022.

[3] Kristen Accardi. Balancing power and performance in the Linux kernel. *LinuxCon, San Francisco*, 2015.

[4] Sugariya Firdous Allaqband, Mir Nazish, Saltanat Firdous Allaqband, Janibul Bashir, and M Tariq Banday. An efficient machine learning based scheduler for heterogeneous multicore processors. *International Journal of Information Technology*, 17(4):2493–2501, 2025.

[5] Zohra Amekraz and Moulay Youssef Hadi. CANFIS: a chaos adaptive neural fuzzy inference system for workload prediction in the cloud. *IEEE Access*, 10:49808–49828, 2022.

[6] Martin F Arlitt and Carey L Williamson. Web server workload characterization: The search for invariants. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):126–137, 1996.

[7] Ganapati Bhat, Gaurav Singla, Ali K. Unver, and Umit Y. Ogras. Algorithmic optimization of thermal and power management for heterogeneous mobile platforms. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):544–557, 2018.

[8] Olivier Bilenne. Dispatching to parallel servers: Solutions of Poisson's equation for first-policy improvement. *Queueing Systems*, 99(3):199–230, 2021.

[9] Dominique Brodowski. CPUFreq governors. Linux kernel documentation, https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt, 2020. `Documentation/cpu-freq/governors.txt` in the Linux kernel source tree. Accessed June 4, 2026.

[10] Xianzhi Cao, Chong Chen, Shiwei Li, Chang Lv, Jiali Li, and Jian Wang. Research on computing task scheduling method for distributed heterogeneous parallel systems. *Scientific Reports*, 15:8937, 2025.

[11] Ellen Cardinaels, Sem C Borst, and Johan SH van Leeuwaarden. Job assignment in large-scale service systems with affinity relations. *Queueing Systems*, 93(3):227–268, 2019.

[12] Jinchao Chen, Pengcheng Han, Ying Zhang, Tao You, and Pengyi Zheng. Scheduling energy consumption-constrained workflows in heterogeneous multi-processor embedded systems. *Journal of Systems Architecture*, 142:102938, 2023.

[13] Jinchao Chen, Yu He, Ying Zhang, Pengcheng Han, and Chenglie Du. Energy-aware scheduling for dependent tasks in heterogeneous multiprocessor systems. *Journal of Systems Architecture*, 129:102598, 2022.

[14] Jing Chen. *Adaptive Task Scheduling and Resource Management Techniques for Improving Energy Efficiency on Multi-core Systems*. PhD thesis, Chalmers Tekniska Högskola (Sweden), 2023.

[15] Jonathan Corbet. What's coming in the next kernel release (part 1). https://lwn.net/Articles/775698/, Dec 2018. Accessed June 2, 2026.

[16] Jonathan Corbet. Completing the eevdf scheduler. https://lwn.net/Articles/969062/, Apr 2024. Accessed June 4, 2026.

[17] Intel Corporation. Intel® speed shift technology. https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/002/intel-speed-shift-technology/, Oct 2021. Accessed June 3, 2026.

[18] Andrew Cunningham. In-depth with the Snapdragon 810's heat problems, Apr 2015.

[19] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 77–88, New York, NY, USA, 2013. Association for Computing Machinery.

[20] Izak Duenyas and Mark P Van Oyen. Heuristic scheduling of parallel heterogeneous queues with set-ups. *Management Science*, 42(6):814–829, 1996.

[21] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. *ACM SIGARCH Computer Architecture News*, 35(2):13–23, 2007.

[22] Dror G Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[23] Adrian Garcia-Garcia, Juan Carlos Saez, and Manuel Prieto-Matias. Contention-aware fair scheduling for asymmetric single-ISA multicore systems. *IEEE Transactions on Computers*, 67(12):1703–1719, 2018.

[24] Kristen Gardner and Cole Stephens. Smart dispatching in heterogeneous systems. *ACM SIGMETRICS Performance Evaluation Review*, 47(2):12–14, 2019.

[25] Annelise Gauthier. Dynamic scheduling-based optimization of multi-core architectures for high-performance computing. *Transactions on Computational and Scientific Methods*, 5(5), 2025.

[26] Gregor Haas, Seetal Potluri, and Aydin Aysu. iTimed: Cache attacks on the Apple A10 Fusion SoC. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 80–90, 2021.

[27] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*, pages 10–14. IEEE, 2014.

[28] Jing Huang, Renfa Li, Jiyao An, Haibo Zeng, and Wanli Chang. A DVFS-weakly dependent energy-efficient scheduling approach for deadline-constrained parallel applications on heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(12):2481–2494, 2021.

[29] The kernel development community. Energy model of devices. Linux kernel documentation, https://docs.kernel.org/power/energy-model.html, 2019. `Documentation/power/energy-model.rst` in the Linux kernel source tree. Accessed June 23, 2026.

[30] The kernel development community. Energy aware scheduling. Linux kernel documentation, https://docs.kernel.org/scheduler/sched-energy.html, 2020. Documentation/scheduler/sched-energy.rst in the Linux kernel source tree. Accessed July 14, 2026.

[31] The kernel development community. Schedutil. Linux kernel documentation, https://docs.kernel.org/scheduler/schedutil.html, 2020. Documentation/scheduler/schedutil.rst in the Linux kernel source tree. Accessed June 5, 2026.

[32] Changdae Kim and Jaehyuk Huh. Exploring the design space of fair scheduling supports for asymmetric multicore systems. *IEEE Transactions on Computers*, 67(8):1136–1152, 2018.

[33] Keqin Li. Optimal task dispatching on multiple heterogeneous multiserver systems with dynamic speed and power management. *IEEE Transactions on Sustainable Computing*, 2(2):167–182, 2017.

[34] Linux Kernel Internals. Scheduler evolution: O(1) → CFS → EEVDF. https://kernel-internals.org/sched/scheduler-evolution/. Accessed June 4, 2026.

[35] Ricardo Mazón and Houcine Hassan. Power-aware scheduling strategies for improving performance and energy consumption in heterogeneous multicore platforms. *The Journal of Supercomputing*, 82(9):524, 2026.

[36] Quentin Perret. Energy Aware Scheduling [patch v8 00/15]. Linux Kernel Mailing List, https://lore.kernel.org/lkml/20181016101513.26919-1-quentin.perret@arm.com/, October 2018. Message-ID: 20181016101513.26919-1-quentin.perret@arm.com. Accessed June 2, 2026.

[37] Aditya Priya, Rajiv Choudhury, Sujay Patni, Himkant Sharma, Moonmoon Mohanty, Krishnasuri Narayanam, Umamaheswari Devi, Pratibha Moogi, Preetam Patil, and Parimal Parag. Energy-minimizing workload splitting and frequency selection for guaranteed performance over heterogeneous cores. In *Proceedings of the 15th ACM International Conference on Future and Sustainable Energy Systems*, pages 308–322, 2024.

[38] Qualcomm Technologies, Inc. Snapdragon 8 Gen 2 mobile platform: Specifications & features, 2022. Product Brief. Accessed: June 2026.

[39] Nikhil Rukmabhatla, Rajshree Chabukswar, Sneha Gohad, and Michael Chynoweth. Intel performance hybrid architecture & software optimizations. *Intel, Tech. Rep.*, 2021.

[40] Bagher Salami, Hamid Noori, and Mahmoud Naghibzadeh. Fairness-aware energy efficient scheduling on heterogeneous multi-core processors. *IEEE Transactions on Computers*, 70(1):72–82, 2020.

[41] Bagher Salami, Hamid Noori, and Mahmoud Naghibzadeh. Online energy-efficient fair scheduling for heterogeneous multi-cores considering shared resource contention. *The Journal of Supercomputing*, 78(6):7729–7748, 2022.

[42] Abul Sarwar. CMOS power consumption and CPD calculation. https://www.ti.com/lit/an/scaa035b/scaa035b.pdf?ts=1784809915937, 1997. Accessed July 24, 2026.

[43] Dominik Schäfer, Janick Edinger, Martin Breitbach, and Christian Becker. Workload partitioning and task migration to reduce response times in heterogeneous computing environments. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–11. IEEE, 2018.

[44] Ashley Stevens. Introduction to AMBA® 4 ACE™ and big.LITTLE™ processing technology. *ARM White Paper, CoreLink Intelligent System IP by ARM*, 2011.

[45] Linus Torvalds. Linux v.7.2, Apr 2026.

[46] Evangelos Vasilakis. An instruction level energy characterization of ARM processors. *Foundation of Research and Technology Hellas, Inst. of Computer Science, Tech. Rep. FORTH-ICS/TR-450*, 2015.

[47] Jiafeng Wang, Sai Xiao, Xiaochuan Guo, Dongfeng Jia, and Wufei Wu. Energy-saving scheduling strategy for real-time applications on big.little architectures. *IEEE Internet of Things Journal*, 12(15):29460–29471, 2025.

[48] Rafael J. Wysocki. intel_pstate CPU performance scaling driver. Linux Kernel v5.3 Administrator's Guide, https://www.kernel.org/doc/html/v5.3/admin-guide/pm/intel_pstate.html, 2019. Accessed June 3, 2026.

[49] Guoqi Xie, Gang Zeng, Xiongren Xiao, Renfa Li, and Keqin Li. Energy-efficient scheduling algorithms for real-time parallel applications on heterogeneous distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3426–3442, 2017.