NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

A Linux Kernel Scheduler Implementation for Asymmetric CPUs

Sergios - Anestis A. Kefalidis

Supervisor: **Alex Delis,** Professor

ATHENS

SEPTEMBER 2022

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μία Υλοποίση Χρονοπρογραμματιστή για Ασύμμετρους Επεξεργαστές στον Πυρήνα Linux

Σέργιος - Ανέστης Α. Κεφαλίδης

Επιβλέπων: **Αλέξης Δελής**, Καθηγητής

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2022

BSc THESIS

A Linux Kernel Scheduler Implementation for Asymmetric CPUs

Sergios - Anestis A. Kefalidis
S.N.: 1115201800073

SUPERVISOR: **Alex Delis,** Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μία Υλοποίση Χρονοπρογραμματιστή για Ασύμμετρους Επεξεργαστές στον Πυρήνα Linux

Σέργιος - Ανέστης Α. Κεφαλίδης
Α.Μ.: 1115201800073

ΕΠΙΒΛΕΠΩΝ: **Αλέξης Δελής,** Καθηγητής

ABSTRACT

Asymmetric CPUs consisting of cores with different processing capacities have been widely used in mobile devices in the last decade. Their design allows chip-makers to offer improved energy efficiency while maintaining solid performance. The other side of the coin is that CPUs of this type require extra care from operating system developers, who have to design schedulers that take advantage of this asymmetry.

Recently, heterogeneous (asymmetric) architectures have started to appear in the PC space, which has traditionally been dominated by homogeneous CPUs. In addition to the energy-efficiency benefits, this development is being propelled by the improved space-efficiency of small cores. Replacing a single big core with several small cores leads to increased multi-threaded performance without increasing manufacturing costs.

Asymmetric Multi-Processing in computers poses some unique challenges that schedulers have to handle. Heterogeneity in computers aims for strong, sustained multithreaded performance in high-load situations and low power consumption in low-load situations. On the contrary, in mobile devices, heterogeneous architectures aim to maximize battery life while offering decent performance to a limited number of tasks with short bursts of CPU-intensive activity. Apart from that, Simultaneous Multi-threading hasn't seen much use in mobile platforms leading to the design of heterogeneity-aware SMT-unaware schedulers. Mishandling SMT can lead to significant performance degradation and is unacceptable for a modern computer scheduler.

In this thesis, we present HCS, a heterogeneity-aware, SMT-aware, general-purpose CPU scheduler for servers, desktops and laptops. It combines utilization-based, bias-based, and fairness-based scheduling schemes to provide efficiency and performance. To add to that, HCS allows both users and application developers to configure its behavior to better suit their needs. HCS is built on top of ULE and has been implemented in the Linux Kernel as a replacement for the energy-aware variant of the CFS scheduler.

SUBJECT AREA: Operating Systems

KEYWORDS: CPU Scheduling, Heterogeneous CPUs, Bias Scheduling, Linux, Kernel Development

ΠΕΡΙΛΗΨΗ

Την τελευταία δεκαετία έχουν χρησιμοποιηθεί ευρέως ασύμμετροι επεξεργαστές που αποτελούνται από πυρήνες διαφορετικών επεξεργαστικών δυνατοτήτων στις κινητές συσκευές. Η σχεδίαση τους επιτρέπει στους κατασκευαστές επεξεργαστών να προσφέρουν ελαττώσουν την κατανάλωση ενέργειας διατηρώντας καλή ταχύτητα. Ταυτοχρόνως, τέτοιοι επεξεργαστές απαιτούν ειδική μεταχείριση από τους προγραμματιστές λειτουργικών συστημάτων, οι οποίοι πρέπει να σχεδιάσουν ειδικούς χρονοπρογραμματιστές που να εκμεταλλεύονται την ασυμμετρία.

Προσφάτως έχουν κυκλοφορήσει ασύμμετροι επεξεργαστές για προσωπικούς υπολογιστές, ένας χώρος στον οποίο παραδοσιακά κυριαρχούν οι συμμετρικοί επεξεργαστές. Αυτή η εξέλιξη είναι αποτέλεσμα της βελτιωμένου ενεργειακού τους προφίλ, καθώς και της χωρικής αποδοτικότητας αυτών των αρχιτεκτονικών. Η αντικατάσταση ενός μεγάλου πυρήνα με πολλούς μικρούς προσφέρει αυξημένη διεκπεραιωτική ικανότητα πολυνηματικών εργασιών χωρίς να αυξάνει το κόστος παραγωγής.

Οι ασύμμετροι επεξεργαστές για υπολογιστές δημιουργούν μία σειρά από προβλήματα που οι χρονοπρογραμματιστές καλούνται να αντιμετωπίσουν. Η ανομοιογένεια στους υπολογιστές αποσκοπεί στον συνδυασμό της ελαχιστοποίησης της κατανάλωση ενέργειας σε καταστάσεις χαμηλού φόρτου και στη μακροχρόνια και ταχύ πολυνηματική απόδοση σε καταστάσεις αυξημένου φόρτου. Αντιθέτως, στις κινητές συσκευές, οι ανομοιογενής αρχιτεκτονικές αποσκοπούν στη μεγιστοποίηση της διάρκειας μπαταρίας, και την γοργή απόδοση σε σύντομα διαστήματα υψηλού φόρτου. Επιπλέον, η ταυτόχρονη πολυνημάτωση (SMT) είναι μία τεχνική που ενώ δεν χρησιμοποιείται σε κινητές συσκευές, χρησιμοποιείται από σχεδόν όλους τους επεξεργαστές υπολογιστών. Η κακή διαχείριση της ταυτόχρονης πολυνημάτωσης μπορεί να οδηγήσει σε σημαντική ελάττωση της αποδοτικότητας του συστήματος.

Στην παρούσα πτυχιακή εργασία παρουσιάζουμε τον HCS, έναν γενικό χρονοπρογραμματιστή για ανομοιογενής επεξεργαστές που υποστηρίζουν ταυτόχρονη πολυνημάτωση. Συνδυάζει υπάρχουσες τεχνικές χρονοπρογραμματισμού ασύμμετρων επεξεργαστών για να προσφέρει ενεργειακή αποδοτικότητα και ταχύτητα. Επιπλέον, ο HCS είναι εύκολο να τροποποιηθεί από τους χρήστες ή τους προγραμματιστές εφαρμογών. Ο HCS εχει υλοποιηθεί, ως επέκταση του ULE, στον πυρήνα των Linux ως αντικαταστάτης του χρόνοπρογραμματιστή CFS.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Λειτουργικά Συστήματα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Χρονοπρογραμματισμός ΚΜΕ, Ανομοιογενής Επεξεργαστές, Χρονοπρογραμματισμός ΚΜΕ με Bias, Linux, Προγραμματισμός Πυρήνα

Στους γονείς και την αδερφή μου, που πάντα με στηρίζουν.

CONTENTS

LIST OF FIGURES

PREFACE

This thesis is part of my undergraduate studies in the Department of Informatics and Telecommunications of the University of Athens. Specifically, this is my last remaining obligation before I complete my bachelor's degree. I want to thank my supervisor, Professor Alex Delis for taking the stress out of creating this thesis, as well as for his invaluable guidance.

1. INTRODUCTION

1.1 CPU Schedulers

The CPU scheduler is one of the core parts of an Operating System. It is responsible for allocating CPU resources to processes (tasks) in a way that achieves the goals of the system designer. In the land of general-purpose schedulers, this means reaching a good compromise between maximizing throughput and minimizing response time.

In single-core systems, all processes share the same CPU. A process should not monopolize a CPU because that would ruin any semblance of fluidity (high response time). To solve that issue, CPU time is divided into slices and each process receives slices depending on some scheduling policy. If the slices are tiny, the system will waste a lot of time switching between tasks (low throughput).

Multicore systems introduce an additional layer of complexity, load balancing between cores. Processes aren't randomly placed on a core and left there until their execution is finished. A multiprocessing-aware scheduler tries to evenly distribute work between all the cores of the system. To that end, the scheduler uses some metric to calculate the load of each core and migrates processes between cores when the load is uneven.

1.2 Heterogeneous CPUs

In the early days of computing, single-core CPUs saw dramatic performance increases every year, leading to lower response times and higher system throughput. As frequency and power requirements increased, chips became harder to cool, making them impractical for personal use. This phenomenon is known as the "Power Wall" and led to the introduction of multicore CPUs, which enabled CPU providers to circumvent thermal issues and increase system throughput [14].

After the introduction of multicore CPUs, designers and researchers started looking for new ways to improve performance, reduce energy consumption, and lower production costs. This is where heterogeneous CPUs come in [11]. There are many types of heterogeneous CPUs, some have cores with different ISAs (functional asymmetry), and others have cores that differ only in their speed (performance asymmetry). The last type has been widely used in mobile devices for many years, one notable example being ARM's big.Little architecture[7].

Many different configurations of cores have been tested, either in research or in products, but the prevailing design has been to use two types of cores [7, 4]. One group consists of small/efficient cores and one group consists of big/performant cores (figure 1.1). Such CPUs enable the OS to use the small/energy-efficient cores for light and/or background tasks, and the big/fast cores for CPU-bound processes. This way, battery performance is improved when the system is under light load without sacrificing system performance under high load.

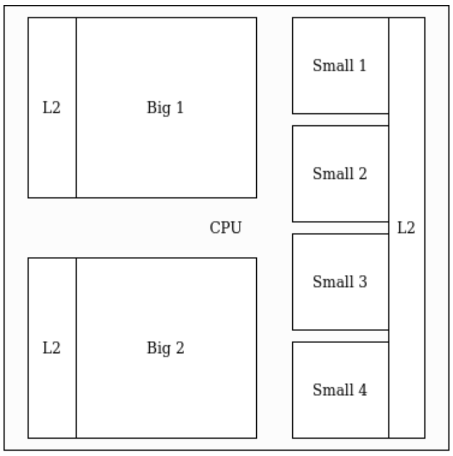


Figure 1.1: An asymmetric CPU consisting of 2 big cores, each of which has its own L2 cache, and 4 small cores, with shared L2 cache

1.3 Schedulers for Heterogeneous CPUs

The main difference between a heterogeneity-aware scheduler (for functional asymmetry) and a conventional scheduler is the ability to decide whether a process should be placed on a strong or on a weak core. Multiprocessing-aware schedulers try to evenly split the load between the cores of the system. This behavior makes sense if all cores are similar, which isn't the case with heterogeneous processors. Instead, the scheduler should try to leverage the asymmetry and place tasks according to the speed and energy efficiency of each core.

Generally, smaller cores are more energy-efficient, so running non-CPU-bound processes on them reduces energy consumption while not impacting response times too much. On the other hand, big cores are less energy-efficient but are able to work faster and are suitable for CPU-bound tasks. The scheduler needs to decide which core-type is the most suitable for each process when a process is created. Additionally, a heterogeneous-aware scheduler must be able to migrate processes from one type of core to another, since both the nature of the process and the system load might change as time progresses.

For example, a cloud syncing application might involve two stages, compressing some data and uploading the compressed data to the cloud. Compression is CPU-bound, and it would probably benefit from using a big core (figure 1.2). On the other hand, uploading doesn't require much from the CPU, so using a small core would probably be more energy efficient (figure 1.3). In this case, a good scheduler would run the compression part of the task in a big core and when the uploading stage is reached, move the task to a small core.

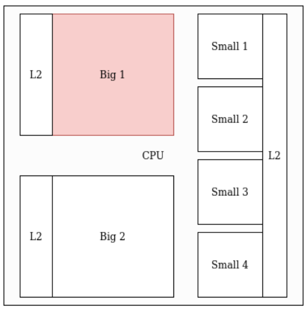


Figure 1.2: Cloud task compressing data on a big core.

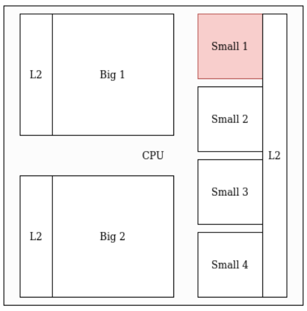


Figure 1.3: Cloud task uploading data on a small core.

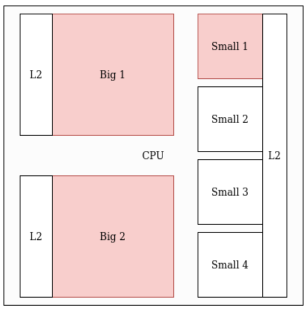


Figure 1.4: Cloud task compressing data on a small core.

In another example, imagine the same cloud syncing application that is compressing data in the background. The user has set a low nice value for this application. As long as there is available capacity in the big cores, this process runs on a big core (figure 1.2). At some point, the user opens a multithreaded competitive video game that requires a lot of CPU-time and has a higher nice value. A good scheduler will run this video game on the big cores, and if there isn't enough CPU capacity for the cloud syncing application, it will be moved to a small core (figure 1.4).

The two previous examples show that heterogeneity in CPU designs is a complex problem that requires purpose-built algorithms to reap its benefits.

1.4 The need for new heterogeneity-aware schedulers

As we mentioned previously, heterogeneous CPU architectures have been used successfully in mobile devices for over ten years. Naturally, CPU schedulers have been developed to handle these platforms. One might wonder whether there is a need to develop new schedulers to handle heterogeneity in computers. The answer to that question is "yes".

One important issue of most, if not all, existing schedulers, is the lack of support for Simultaneous Multi-threading (an important technique used by x86 CPU designers, but not in mobile platforms). Another important issue that plagues some existing schedulers is the inability to effectively schedule systems at both low and high system loads. Finally, most heterogeneity-aware schedulers are unfair, which is expected on low systems loads but might be undesirable for systems under high load.

In this thesis, we make the case for HCS, a performant, energy-efficient, SMT-aware, and easily configurable scheduler for asymmetric CPUs that combines utilization, bias, and proportional-share mechanisms (fair [8]).

The rest of this thesis is organized as follows. In chapter 2 we present previous work done on scheduling for heterogeneous CPUs. In chapter 3 we present HCS in detail and in chapter 4 we document the implementation and validation of HCS.

2. RELATED WORK

2.1 Overview

Scheduler design has always been an area of compromise, a conventional scheduler can't maximize both response time and throughput. Likewise, a heterogeneity-aware scheduler can't maximize both performance (throughput and response time) and energy efficiency. A good scheduler for asymmetric processors has two objectives:

1. to be energy efficient, without noticeably sacrificing performance in low-medium load situations.
2. to maximize performance in high load situations.

To achieve these objectives, the scheduler requires information about each process. Information can either be prepared offline, in the compilation or development stages, and provided to the scheduler as part of the executable program, or gathered in real-time. There are two main techniques that fall under the latter category:

1. Choosing the best core-type for each process by watching its CPU utilization, meaning the amount of time it (i.e. the process) is running or wants to be running. This way, we can separate CPU-bound processes from IO-bound processes and place them appropriately [1, 2, 7].
2. Reading performance counters, usually provided by a performance monitoring unit (PMU), and calculating a bias score for each process. This bias score is subsequently used to determine the most suitable core-type. Some widely used performance counters are *last-level-cache misses*, *retired instructions per second*, and *page faults*. The goal is to ensure that the big cores are used effectively instead of wasting their cycles waiting for other parts of the system [9, 13, 17, 18].

2.2 Utilization-based Scheduling

Utilization-based approaches have seen widespread success in the real world, specifically in the Linux Kernel as extensions to CFS (Completely Fair Scheduler). Linux has been at the forefront of developing asymmetric CPU schedulers, thanks to its role in the Android operating system.

2.2.1 Capacity Aware Scheduling

The goal of Capacity Aware Scheduling (CAS) [1] is to provide every process in the system with as much CPU-capacity as it requires, as long as that is possible. It doesn't try to utilize the architectural asymmetry to optimize power consumption, instead, it tries to optimize throughput. As stated in its name, the main pillar of this scheduler is the notion of CPU Capacity, which is a MIPS-like metric of CPU performance. Capacity describes the maximum throughput of each core-type. It isn't a value that is calculated by the scheduler, instead, it is provided by the CPU manufacturers, and it is normalized against the fastest core in the system.

For capacity-aware scheduling to work, the scheduler needs a metric to represent the amount of CPU capacity that a task requires. In CFS, this is called ‘task utilization’. Frequency and architectural differences must be taken into account when calculating task utilization. One second on a fast core isn’t the same as one second on a slow core. For that reason, the following equation is used to make task utilization core and frequency invariant:

$$task\_util(p) = duty\_cycle(p) * \frac{current\_frequency(CPU)}{max\_frequency(CPU)} * \frac{capacity(CPU)}{max\_capacity}$$

Now that we have both a capacity value for every core and the ability to compute the amount of capacity that each task requires (using the previous formula), the scheduler needs to make sure that each task is placed on a core that has enough capacity. For new tasks, the scheduler estimates the required capacity and places the task on a core with space capacity. If the guess proves to be incorrect or if a process at some point of its execution requires more CPU capacity than is available in its current core, it is moved to a core with enough spare capacity.

Capacity, as a metric, can’t accurately describe the relative performance of different core architectures. There is no guarantee that because task X is 10% faster when running on a big core compared to running on a small core, task Y will see the same speedup. That’s an important limitation of all scheduling methods that use Capacity [1].

2.2.2 Energy Aware Scheduling

Energy Aware Scheduling (EAS) [2] is an evolution of Capacity Aware Scheduling (CAS). Instead of only trying to optimize throughput, energy-aware scheduling tries to find the optimal compromise between throughput and energy efficiency.

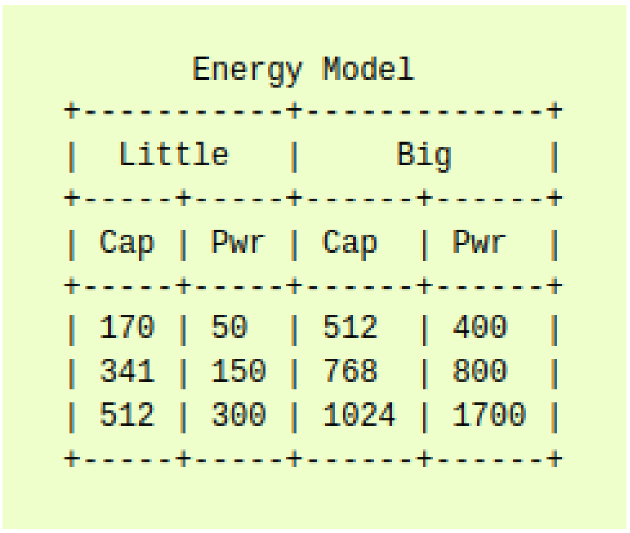
For EAS to work, an energy model of the CPU is required. This energy model contains information about the power consumption of each core-type at different capacity levels (meaning, at different frequencies, provided that the core supports frequency scaling). Like Capacity, the energy model is provided by the CPU manufacturer (figure 2.1).

The scheduler uses this information to select the core that will increase total power consumption the *least*, among the cores that have adequate capacity for the process that is being scheduled. Generally, this means that processes are scheduled to smaller cores if they have enough available capacity, but in some cases, small cores are less efficient at high frequencies than big cores at low frequencies.

Energy-aware scheduling is effective when the system is under light or moderate load, but it is unable to handle high loads, since it doesn’t have any mechanisms to take advantage of asymmetry when all cores are loaded and the system is running at its maximum capacity. For that reason, when a core is used at more than 80% capacity, the system disables the energy-aware mechanisms, and the regular CFS load balancing algorithm is used. Like most schedulers for asymmetric CPUs, EAS doesn’t support SMT (simultaneous multithreading), instead treating sibling cores like physical cores.

2.2.3 Global Task Scheduling

One other notable scheduler is ARM’s Global Task Scheduler [7]. This scheduler, which is also built on top of CFS, moves processes between big and small cores depending on



Energy Model

Little		Big	
Cap	Pwr	Cap	Pwr
170	50	512	400
341	150	768	800
512	300	1024	1700

Figure 2.1: Example of an Energy Model, taken from the Linux Kernel documentation. Different capacity levels for the same CPU are essentially different frequencies of operation.

their CPU demands (tracked using PELT [3]).

Every new process is always placed on a big core. If it turns out to not require much CPU, i.e. task utilization falls below the down-migration threshold set by the developer, it is then moved to a small core, otherwise, it stays on the big core. The opposite happens if a process is placed on a small core and its computational demands increase, i.e. task utilization rises above the up-migration threshold.

When the system is under full load, tasks are periodically moved to the small cores to improve the total throughput of the system, disregarding the thresholds.

2.3 Bias Scheduling

Bias scheduling has been studied extensively over the years. A bias scheduler calculates a bias value for each process, which describes how suitable each core-type is for the process [9, 13, 17, 18].

For example, imagine a process that never sleeps, demanding 100% of a core's capacity. Utilization-based schedulers would place it on a big core. The question now is what happens if the process spends half of its runtime waiting because the cache cannot handle its demands. Our memory-intensive process could be moved to a small core to make space for other compute-intensive processes that effectively utilize the big core's CPU time, thus increasing system throughput. Essentially, bias schedulers place tasks based on how ”well” they spent CPU-time, while utilization-based schedulers place tasks based on how much CPU-time they require.

In [9] researchers used IPC (instructions per cycle) ratios between core-types (2.1) as the bias. In this case, a task that has a bigger bias score than another task benefits more from running on a big core. The end goal is to maximize the total IPC. To achieve that, the tasks with the biggest IPC ratios are placed on the big cores and the rest are placed on the small cores.

$$IPC\_Ratio = \frac{IPC\_in\_big\_core}{IPC\_in\_small\_core} \tag{2.1}$$

For this to work, the scheduler needs to know the IPC value of each process on each core type. To that end, processes periodically change core-type, to update the information used to calculate the IPC ratio. Processes that exhibit IPC fluctuations are swapped between core types independently, to update their IPC ratios and be appropriately scheduled. Additionally, there is a minimum time spent in each core, to ensure that processes are not constantly being moved.

In [13], the authors use a variety of metrics (retired instructions, cache-misses, tlb-misses) to calculate process bias. Subsequently, tasks are placed in the appropriate core-type by modifying the load-balancing algorithm of CFS.

In [17], performance monitoring counters are used to compute a speed-up factor for each process, using an additive-regression engine, which is then used to determine which processes would be best suited for execution on big cores.

The main disadvantage of bias schedulers is that they can lead to inefficient task placement in low or medium system loads. For example, a bias scheduler might place a task with a very light, periodic activity on a big core because it uses big cores effectively. The problem, in this case, is that this task could be running on a small core with no noticeable difference in response time since it has a light CPU-load, but since it is placed on a big core, it doesn't allow the big core to become idle to reduce its power consumption.

2.4 Fairness

Fairness is a key concept in homogeneous schedulers. The two most successful open source schedulers (CFS and ULE) are both fair schedulers. Fairness means splitting resources between tasks in such a way that all tasks receive some amount of CPU-time depending on their priority and behavior [8]. Such designs allow all tasks to progress, and seem to provide the best combination of throughput and responsiveness.

The heterogeneous schedulers that were presented previously are all inherently unfair. Some tasks are placed on big, fast cores, while other tasks are placed on small, slower cores. Both bias and utilization-driven scheduling methods aim to leverage the asymmetry of the system, which means that tasks that are deemed unsuitable for big cores never have a chance to run on them.

This way of thinking might prove problematic in a system that has heavy and heavier tasks. All tasks would benefit from running on the big cores, but the heaviest tasks in the system monopolize the big cores. In this case, it can be beneficial to give the "less CPU-bound" tasks the opportunity to run on the big cores, essentially sharing big-cores depending on task utilization and/or bias [15].

There have been attempts to create fair schedulers for asymmetric systems [15, 18]. In [15], researchers combined bias scheduling with lottery scheduling to share (proportional-share) the big cores between tasks and showcased promising results, albeit in a limited environment. In [18] a fair scheduler that trades throughput for fairness, by swapping tasks between cores depending on a runtime metric, is presented, built upon [17] and Linux's CFS.

2.5 Offline Profiling and Signature Matching (HAAS)

A completely different approach to scheduling asymmetric CPUs was proposed in [19]. The Heterogeneity-Aware Signature Supported scheduler uses architectural signatures to estimate the completion time of the application on each core-type. Depending on that estimated completion time, processes are placed in the appropriate cores and stay there until their execution finishes. An architectural signature contains information about the process, specifically about its cache access patterns. The signatures are built together with the executable, possibly as an additional compilation step. This is an example of a scheduler that doesn't use online monitoring.

This approach trades accuracy and flexibility for reduced overhead and scalability. One downside of this approach is that because each application has one architectural signature, the scheduler can't handle applications with varying demands on CPU resources. Another significant downside is that it can't handle applications with significantly different behavior depending on user input, again because each application has only one signature.

2.6 Summary

We presented the main approaches to asymmetric CPU scheduling. Each approach has a unique set of advantages and disadvantages. Utilization-based algorithms are not great at scheduling systems under high load, bias and fair schedulers may lead to suboptimal scheduling for systems under low or medium loads. In addition, none of the aforementioned schedulers supports SMT. For HCS, we combined all these techniques to create a scheduler that can handle low, medium, and high loads as well as architectures that utilize SMT.

3. HCS: HETEROGENEOUS CPU SCHEDULER

The Heterogeneous CPU Scheduler is an attempt to create a robust scheduler that is capable of utilizing system resources effectively on a variety of systems and system loads, providing both energy efficiency and performance. To achieve that, we combined utilization, bias and proportional-share (fair) scheduling. It is important to note that HCS is only tasked with placing tasks on core-groups (we refer to all cores of the same type in a system as a core-group). Each core-group is scheduled by a conventional, multiprocessor scheduler, in our case ULE [16].

HCS uses utilization-based and bias mechanisms for low to medium system loads, and bias and proportional sharing mechanisms for high system loads.

3.1 Scheduling Low - Medium System Load

A system under low to medium load has enough CPU capacity for all running tasks, so system throughput isn't a concern. Instead, the focus is on energy efficiency and responsiveness. The utilization-based mechanisms make sure to avoid placing light tasks on big cores. At the same time, the use of bias allows us to avoid placing tasks with medium CPU requirements on big cores, if they have a high bias value (which means that the task is better suited for a small core). This part of HCS is an evolution of Global Task Scheduling proposed in [7]. The most significant improvements are the use of bias and dynamic migration thresholds.

3.1.1 HCS Score

The scheduler calculates the task utilization (CPU load) and bias (core-type suitability) of each process. These two values are combined into a single value called HCS Score (abbreviated to HScore). In this subsection, we will present how each score is calculated and how they are combined into the HScore.

For task utilization, HCS uses ULE's Interactivity metric [16], which categorizes processes as IO or CPU bound in varying degrees on a range of 0 (IO-bound) to 100 (CPU-bound). A process that voluntarily gives up CPU time is classified as IO-bound, while a process that uses most/all of its CPU slices is classified as CPU-bound. The following set of equations calculate the interactivity value of a process:

$$Interactivity = \begin{cases} \frac{m}{\frac{sleep}{run}} & \text{for } sleep > run \\ \frac{m}{\frac{run}{sleep}} + m & \text{for } sleep \leq run \end{cases} \tag{3.1}$$

$$m = \frac{MaxInteractiveScore}{2} = 50$$

Sleep is the number of ticks that have passed between a *sleep()* and *wakeup()* or while sleeping on a condition variable. The runtime (run) is the number of ticks that the thread has been running. Neither sleep-time nor runtime are allowed to grow infinitely, instead being reduced to a fraction of their size when their sum reaches a configurable limit. This essentially limits the history kept.

The careful reader might notice a problem with this Interactivity formula, in the context of asymmetric processors. Cores of different types have different performance profiles, so interactivity values between cores aren't comparable. To deal with that, we must make the runtime (run) core invariable, as it is done in CFS:

$$InvariantRuntime = Runtime * \frac{capacity(current\_CPU)}{max\_capacity} \tag{3.2}$$

To calculate bias, HCS uses *Last Level Cache Misses*, *Data-TLB misses* and *Instruction-TLB misses*, which are tracked via hardware performance counters. A configurable amount of history is kept for each value to avoid bias fluctuations that don't represent the general behavior of an application. To set up and read these counters, we used *perf*, a Linux Kernel subsystem for performance monitoring and analysis. Bias is constrained to a range of 0 to 100.

Again, architectural differences between cores might cause bias calculations to differ between core types. Currently, there isn't a system in place for HCS to make bias core-type invariant. A possible solution to this problem was presented in [13].

By putting these two values together, we calculate the HScore of a process *p*:

$$HScore(p) = Interactivity(p) - Bias(p) * W \tag{3.3}$$

In this equation, *W* is a configurable weight that reflects the importance of Bias in the system. The default value for W is 0.5 and the full range of possible values is [0, 1].

We chose to subtract bias from interactivity, instead of adding the values, because it is advantageous to move a medium-load task that ineffectively spends CPU time away from a big core, to a small core. At the same time, it doesn't make much sense to migrate a process that doesn't use much CPU time to a big core just because it has a low bias score. HScore expresses both the amount of CPU capacity that a task needs and how well the task uses the CPU-time that is given to it. A task with a large HScore should be placed on a big core and a task with a low HScore should be placed on a small core.

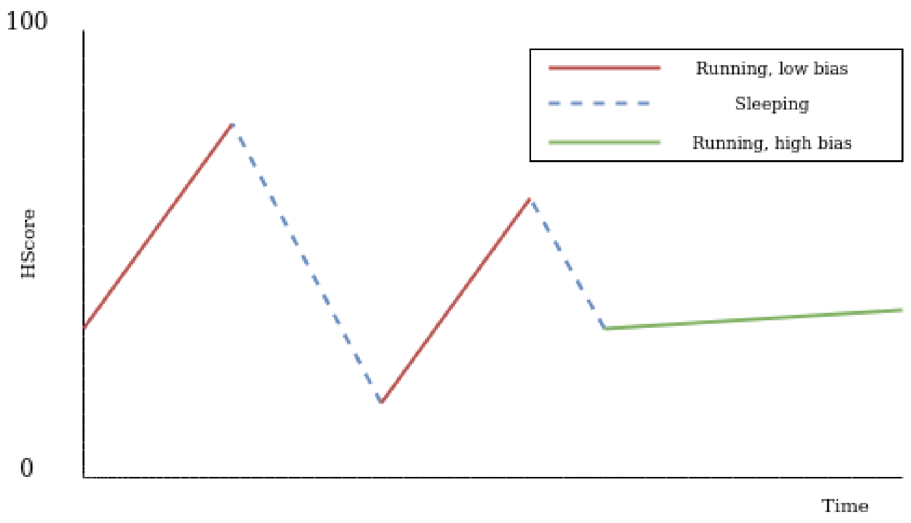


Figure 3.1: A task's HScore graph.

Figure 3.1 shows a graph of the HScore of a sample process. When the process is running and its bias is low, its HScore increases rapidly. When the process sleeps, the HScore decreases, because the Interactivity value decreases. In the end, the process is running but because its workload is cache and/or TLB intensive, its HScore rises slowly, because simultaneously with the increasing interactivity the bias value is increased.

We explained how HCS calculates the HScore of each task. In the next subsections, we explain how HCS uses the HScore to decide on which core-group to place each task.

3.1.2 Wake Up Migration

Each core-group "owns" a range of HScore values. At system startup, the big cores own the range [30-100] and the small cores the range [1-30], with 30 being considered the "border". Additionally, there are two dynamic thresholds, one above the border and one below the border, used for big-core (up) migration and small-core (down) migration respectively, with initial values of 35 and 25 (an offset of 5 from the border). When the border is moved (see section 3.1.5), the thresholds are moved with it, maintaining the offset.

Wake up migration happens when a task becomes runnable and the scheduler has to decide on which core to place it. Normally, tasks are placed on the core that they were running on before they slept or a core near it. In our case, this event is a good opportunity to move tasks to the correct core-group. If a task's HScore right before sleeping is at or under the small-core migration threshold, the task is placed on a small core (figure 3.2). We don't use the current HScore because that would move CPU-bound tasks that voluntary sleep for a long time to the small-core group, only to then immediately move them back to the big-core group.

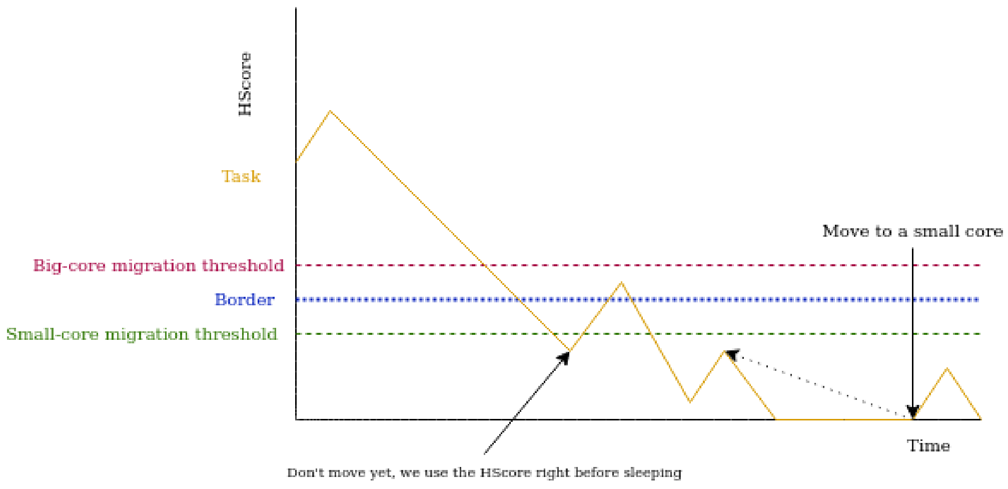

HScore
Task
Move to a small core
Big-core migration threshold
Border
Small-core migration threshold
Time
Don't move yet, we use the HScore right before sleeping

Figure 3.2: Wake Up Migration.

3.1.3 Active Migration

If a task never goes to sleep voluntarily and is running on a small core, Wake Up Migration is unable to move the task in question to a big core. This is where Active Migration comes in. A running task is moved to a big core as soon as it reaches the up-migration threshold (figure 3.3), instead of waiting to sleep. Active Migration only supports moving tasks to the big cores, moving tasks to small cores while they are running is unlikely to improve system performance, and it is something that only the load balancer can do if it is deemed necessary.

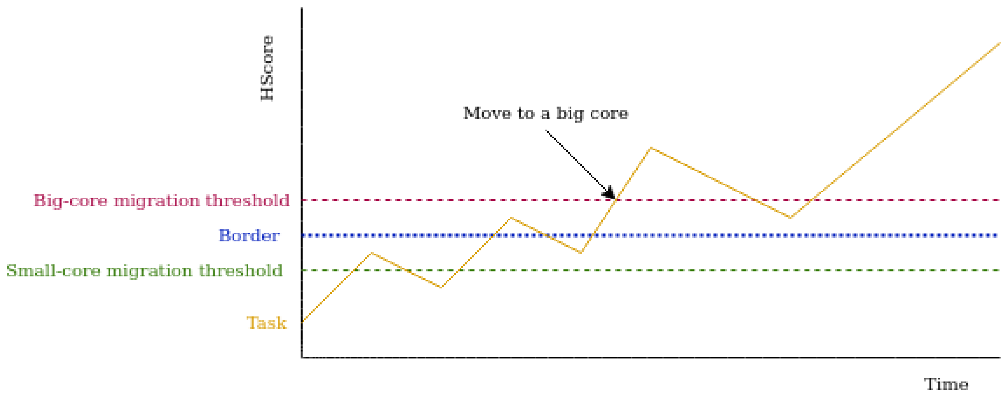


Figure 3.3: Active Migration.

3.1.4 Forking - New Process Placement

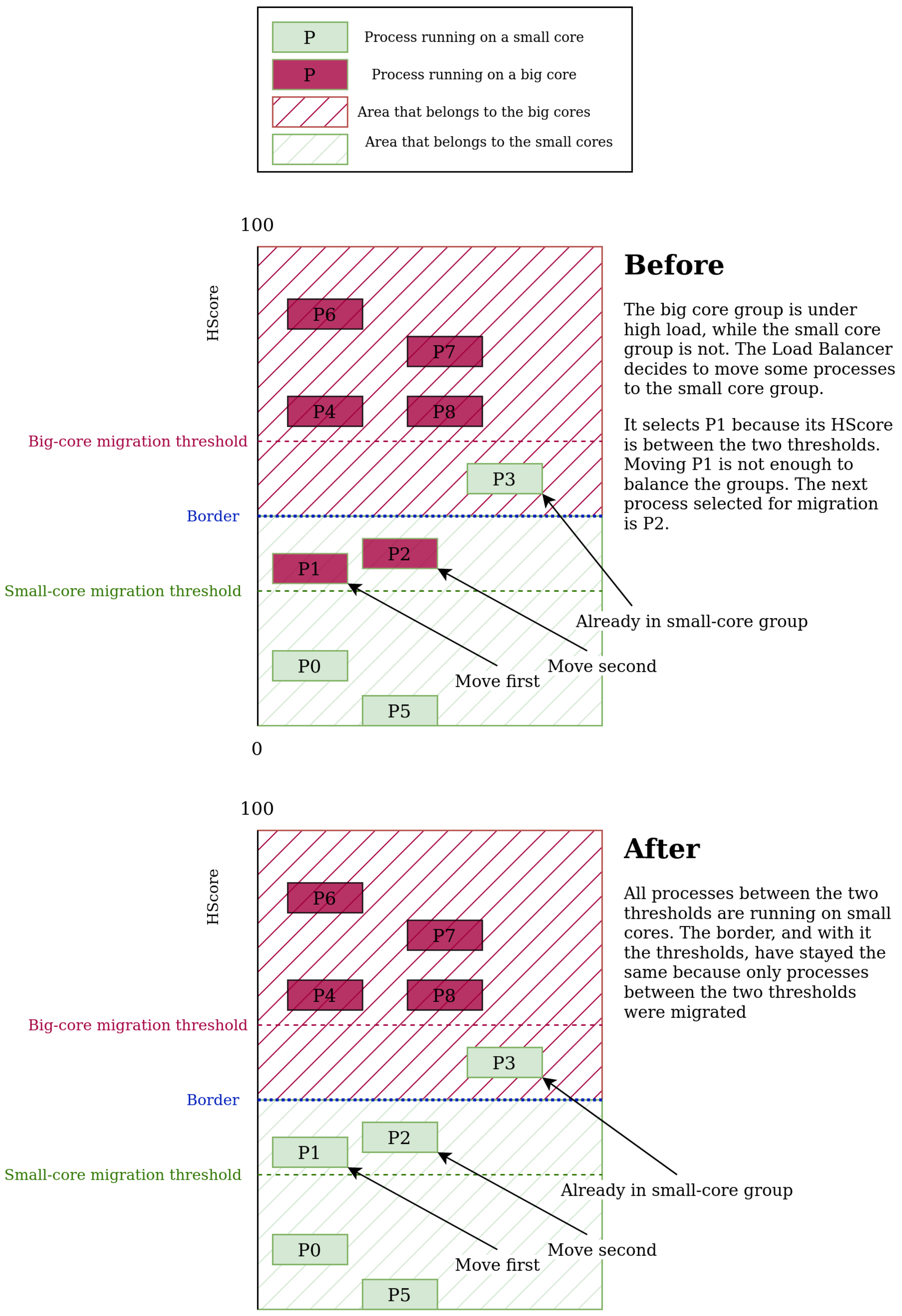
All new processes are placed on big cores, provided that there is adequate space for them. Otherwise, they are placed in the small core-group. If neither core-group has enough space for the new process, then the system is under high load. If a process is incorrectly placed on a big core, it will quickly migrate to a small core. This design choice trades a bit of energy efficiency, for responsiveness, which is a good trait for a general, multipurpose scheduler.

3.1.5 Load Balancing

The Load Balancer runs periodically and checks the load of the core-groups (the mean of the loads of each core in the group). If neither core-group is under “high load” (meaning more than 80% utilization) then nothing happens. When one of the two core-groups is under high load, we want to move tasks to the other core group, to better balance the load of the two core groups. The goal of load balancing is to avoid overwhelming a single core-group to keep the utilization-based mechanisms working as well as optimizing throughput and responsiveness.

At first, HCS attempts to move tasks away from the overloaded group, choosing from the ones that exist between the two migration thresholds. Depending on the load and spare capacity of the core-groups, tasks are moved until no core-group is overloaded anymore or until both core-groups are overloaded (figure 3.4). This process doesn’t modify the border and thresholds.

If there aren’t enough tasks in the zone between the two thresholds to achieve balance and one core-group continues to be overloaded while the other is not, HCS migrates tasks that are outside the “zone”. The difference here is that once the load balancer finishes, it adjusts the “owned” HScore ranges of each core-group to better reflect the state of the system (figure 3.5). This means that the border and the migration thresholds are moved, which makes it harder to reach the same unbalanced state if the system continues to be used in a similar way. Subsequent executions of the Load Balancer check to see if the system is under light load. If so, the border and the migration thresholds are returned to their original locations. By moving the thresholds via the Load Balancer, HCS avoids relying on the load balancer to schedule medium-load situations with tasks that don’t fit under the default migration thresholds. Instead of constantly needing the Load Balancer to move tasks from the overloaded core-group to the other one, tasks are naturally moved through Active and Wake Up Migration, reducing the overhead of HCS.


P
Process running on a small core
P
Process running on a big core
Area that belongs to the big cores
Area that belongs to the small cores
100
HScore
P6
P7
P4
P8
Big-core migration threshold
P3
Border
P2
P1
Small-core migration threshold
Already in small-core group
Move second
Move first
P0
P5
0
100
HScore
P6
P7
P4
P8
Big-core migration threshold
P3
Border
P2
P1
Small-core migration threshold
Already in small-core group
Move second
Move first
P0
P5

Before

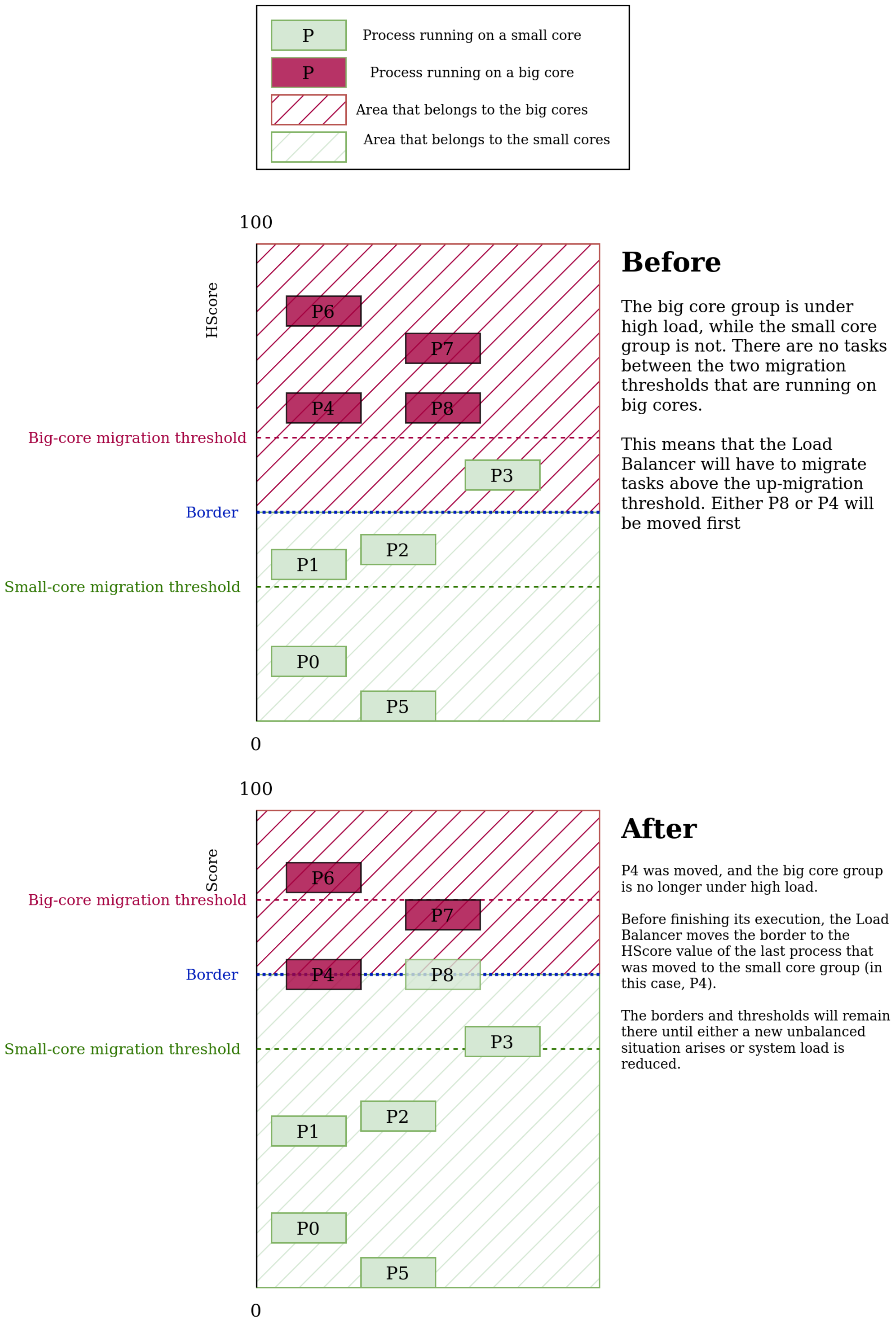
The big core group is under high load, while the small core group is not. The Load Balancer decides to move some processes to the small core group.

It selects P1 because its HScore is between the two thresholds. Moving P1 is not enough to balance the groups. The next process selected for migration is P2.

After

All processes between the two thresholds are running on small cores. The border, and with it the thresholds, have stayed the same because only processes between the two thresholds were migrated

Figure 3.4: Load Balancing by migrating processes between the migration thresholds.


P
Process running on a small core
P
Process running on a big core
Area that belongs to the big cores
Area that belongs to the small cores
100
HScore
P6
P7
P4
P8
Big-core migration threshold
P3
Border
P2
P1
Small-core migration threshold
P0
P5
0
100
Score
P6
Big-core migration threshold
P7
Border
P4
P8
Small-core migration threshold
P3
P2
P1
P0
P5
0

Before

The big core group is under high load, while the small core group is not. There are no tasks between the two migration thresholds that are running on big cores.

This means that the Load Balancer will have to migrate tasks above the up-migration threshold. Either P8 or P4 will be moved first

After

P4 was moved, and the big core group is no longer under high load.

Before finishing its execution, the Load Balancer moves the border to the HScore value of the last process that was moved to the small core group (in this case, P4).

The borders and thresholds will remain there until either a new unbalanced situation arises or system load is reduced.

Figure 3.5: Load Balancing by migrating processes outside the migration thresholds.

3.2 Scheduling High System Load

A system under high load (both core groups are utilized over 90%) focuses on throughput. As we discussed previously, high load neutralizes utilization-based mechanisms. For example, Linux disables its Energy Aware Scheduler when one core reaches over 80% utilization and reverts to the standard CFS load balancer. On the other hand, pure bias scheduling can lead to performance improvements compared to random scheduling, but it also leads to some tasks monopolizing the big cores, which has been proven to be harmful to system performance in [15, 18]. Instead, HCS uses bias to fairly share big cores between processes.

3.2.1 Proportional-share Mechanism

The moment a system first becomes highly loaded, the Load Balancer attempts to evenly split the load between all cores in the system. Subsequently, the normal load balancing and inter-group migration mechanisms are disabled in favor of the Proportional-share Mechanism (PSM). When system load decreases and the system stops being highly loaded (both groups fall under 80% of utilization), PSM is disabled and the normal load balancing and migration mechanisms are re-enabled. The small difference between the threshold for enabling and disabling PSM exists to ensure that the scheduler doesn't constantly switch between the two types of scheduling, which is highly unlikely to begin with.

PSM is a lottery scheduling algorithm for heterogeneous CPUs, in the same vein as [15]. Lottery schedulers distribute tickets to each process in the system and decide which processes will be scheduled next by randomly selecting tickets [8]. PSM applies this logic to migration to big cores. The selected process is moved to a big core, if it is not are not already on one. At the same time, the last process that ran on that big core is moved to the small core from which the lucky process originated. PSM uses the bias score of each process to calculate the number of tickets to give to each process. The smallest the bias (more effective big-core utilization) the larger the number of tickets. The amount of tickets selected each time the lottery is run depends on the core configuration. The amount is equal to the number of cores in the smallest core group. So, in a system with 8 big cores and 4 small cores, 4 tickets will be chosen each time the lottery runs.

Until the next lottery execution, the scheduler watches the swapped tasks and calculates system speedup/slowdown by calculating the IPC ratios of the swapped tasks, similarly to [9]. If the system was sped-up, the task that was placed on the big core receives a decaying number of bonus tickets (the bonus starts at +50% and is reduced by 10% for every lottery execution). PSM runs every 3 - 5 seconds, the exact number is randomly chosen.

Both the ticket bonus and the frequency of the lottery are configurable. More frequent lottery execution improves system fairness, while running the lottery less often increases throughput, provided that the bias score is working well for the system's workload.

To minimize the overhead of the lottery algorithm, a delayed-lottery implementation was chosen for HCS. Instead of searching for the process with the winning ticket, HCS does the following:

1. Find the core with the winning ticket. This is simple since PSM knows the number of tickets on each core (they are updated every time the bias calculations are done).
2. If that core is a big core, do nothing. Alternatively, tell the small core to find the

winning ticket by giving it the winning ticket and suspend the lottery scheduler until the process is found.

3. For every process that is executed on the small core, subtract its ticket number from the winning ticket number.
4. The process that has more tickets than the winning ticket number is the winner and will be moved to the queue of a big core by the small core.

This implementation introduces a negligible overhead both computationally, an additional subtraction is carried out on each context switch, and memory-wise, no special data structure is needed. The main disadvantage of this implementation is that the completion time of the lottery algorithm varies depending on the number of processes in the system and the selected ticket numbers, although a system that has enough CPU-bound processes for this to be a problem has bigger issues to deal with.

3.2.2 Forking - New Process Placement

When a system is under high load, new processes are placed on the core with the least amount of load.

3.3 Simultaneous Multi-threading

Simultaneous Multi-threading (SMT) is a technique that allows a single core to run multiple threads in parallel [14]. The implication for schedulers is that the number of logical, schedulable cores isn't the same as the number of physical cores (e.g. a dual-core system with SMT has four logical cores). Sibling cores, logical cores that co-exist in the same physical core, share the execution components and cache of the core. As one might imagine, this makes logical cores weaker than physical cores. For this reason, most conventional schedulers for homogeneous CPUs handle SMT by having only one runqueue per physical core, with logical cores sharing processes (the migration penalty between sibling cores is minuscule).

In HCS, the existence of SMT poses two questions:

1. How do sibling cores affect core and group utilization? For example, if a core is utilized at 80% and its sibling core at 20%, this doesn't mean that half of its processing power is used. Instead, the core is being used at approximately 75% of its total capacity (depending on the SMT implementation).
2. How do sibling cores affect the Interactivity metric? Running two CPU-bound processes on the same physical core is significantly slower than running them on two different physical cores, and this is reflected in the Interactivity score of the processes.

In both light-to-medium and high-load scheduling, HCS makes decisions related to core utilization on a physical core basis. This means that a dual-core system with SMT has two cores as far as load balancing in HCS is concerned. Additionally, HCS uses the busiest of the sibling cores as a representation of the utilization of the physical core. For example, if CPU0 and CPU1 are sibling logical cores, and CPU0 is utilized at 30% while CPU1 is utilized at 40%, the physical core is utilized at 40% as far as HCS is concerned.

As far as the Interactivity Score of each process goes, in low and medium loads, there is more than enough CPU capacity for all processes in the system, so SMT doesn't noticeably affect Interactivity. Any good SMT-aware scheduler makes sure to avoid putting tasks on the same physical cores if there is another physical core available, and there always is by definition in such loads. In high-load situations, interactivity doesn't matter since utilization-based mechanisms are disabled in favor of PSM.

3.4 Niceness

HCS uses nice values to break ties when load balancing. If two tasks have the same HScore their niceness (nice value) is used to decide which one to place on a big core and which one to place on a small core (figure 3.5).

3.5 Quality of Service

One last feature of HCS, is the inclusion of a nice-like Quality of Service (QoS) value, inspired by Apple's scheduler [12, 6]. There are two classes of service, *normal* and *background*. All tasks start in the *normal* class, but the user or the task itself can decide to set QoS to *background*, permanently or temporarily. A *background* task is always scheduled on a small core, while a *normal* task can be scheduled on either core type depending on its behavior. This feature is useful for minimizing the effect that long-running CPU-bound tasks (e.g. compressed system backup) have on system performance.

4. IMPLEMENTATION AND VALIDATION

In this chapter, we discuss our implementation [5] of HCS in the Linux Kernel and present some tests to validate it. Due to the lack of the required hardware, there are no benchmarks to present in this thesis.

4.1 Implementation

HCS is a heterogeneity-aware scheduler that handles placing tasks on the correct core-group (a core-group is a group of architecturally identical cores). Scheduling cores of the same type is left to a conventional, heterogeneity-unaware scheduler. For our implementation, we use ULE, the FreeBSD scheduler, as our intra-core-group scheduler. We chose ULE for its simplicity (around 3,000 lines of code) compared to CFS (over 10,000 lines of code) which made it easier to develop HCS, without sacrificing performance [10].

Implementing HCS in the Linux Kernel was done in four steps:

1. Update the ULE patch created in [10] to work with the latest version of the Kernel.
2. Setup *perf* to collect the counters needed for calculating the bias of each process.
3. Setup core utilization tracking.
4. Implement the HCS algorithm on top of ULE.

4.1.1 Updating ULE to work on the latest Linux Kernel

The end goal of this work is to build a system with an asymmetric desktop CPU and compare HCS and CFS (both EAS and CAS). For our comparison to be fair, we want CFS to have all the improvements that have been introduced in the years since the creation of the ULE port for Linux. To enable that, we rebased (git rebase) HCS on the latest, at the time of development, version of the Linux Kernel. The kernel has undergone a significant amount of restructuring since 2018, so one contribution of this work is a working version of ULE for Linux 5.19.

4.1.2 Collecting performance counters for all processes

We decided to use the *perf* subsystem of the Linux Kernel to read hardware performance counters. *Perf* is mostly used through the userspace application of the same name, examples of in-kernel usage of it are hard to find. Fortunately, the NMI Watchdog uses *perf* to detect if the system has locked up. Based on that, we set up performance monitoring via the function *perf_event_create_kernel_counter()*.

In our earliest attempts, we tried to set up and read performance counters inside the scheduler code. Unfortunately, this proved to be impossible because scheduler code is non-preemptible, while the functions to interact with *perf* in the Kernel require preemption. In the end, we decided to set up process monitoring on process creation, in *kernel_clone()*. This way, performance counters are set up when a process is created and the counters are updated when a process is running, at a set frequency, and outside the scheduler code.

4.1.3 Core utilization

The original ULE port didn't include setting up core utilization for the scheduler. In other words, the Kernel was unaware of the amount of CPU utilized by each core for tasks being scheduled via ULE. This meant that subsystems that depend on that information (e.g. CPUFreq Governor) would misbehave, even though this was unimportant for the scope that the port was originally created for.

HCS needs per-core utilization information for its load balancing algorithm, so we had to implement that. We decided to use PELT[3] instead of other utilization tracking mechanisms (like WALT, suggested by Qualcomm) to make it easier to compare HCS with CFS.

4.1.4 Implementing HCS

Implementing the HCS algorithm itself was the most straightforward part of the implementation. HCS uses ULE's data structures, and its logic neatly fits on top of the *SCHEDULER_CLASS* interface that Linux uses for its schedulers. It is important to note that our implementation of HCS in the Kernel is a work in progress. Some features, namely SMT support, Niceness and Quality of Service, haven't been implemented yet.

4.2 Validation

Lacking the required hardware to present benchmarks, we present and discuss some of the validation tests that we use for developing HCS. The validation tests that we will present are targeted at the utilization and bias based mechanisms of HCS (presented in 3.1). *PSM* provides fairness, so there isn't anything meaningful to showcase using our simulated environment. All programs that we have created and use for validation are included in the git repository of HCS[5], in the *sources* folder.

Figure 4.1 presents a graph of the execution of *hcs_primes_burst*, a CPU-bound program that calculates a large number of prime numbers. The process starts on a big core and as it never voluntarily sleeps, its HScore increases until it reaches the maximum value. This application has a low bias score (meaning that it is suitable for big cores), so HScore is primarily influenced by the Interactivity Score.

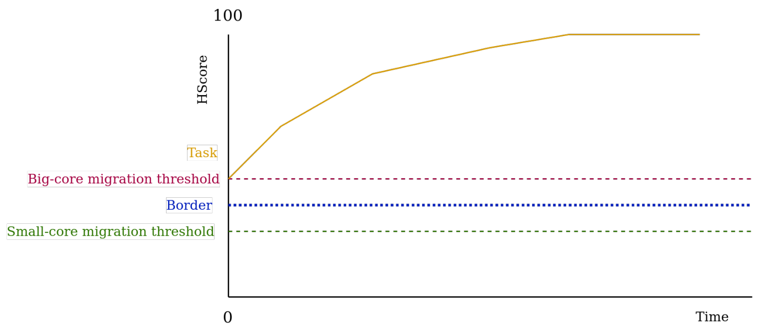


Figure 4.1: Execution of hcs_primes_burst.

Figure 4.2 is the execution graph of *hcs_primes_periodic*, which calculates a small number of prime numbers every few seconds. This is an example of a periodic process with light computational demands, like a chat application or an email client. Again, this process has a low bias value, but because its processing needs are relatively minor it is more efficient to place it on a small core. Initially, the process is placed on a big core, it subsequently sleeps for a long period of time, but when it wakes up it stays on the big core. The second time it wakes up, HCS understands that this process is better suited to run on a small core, and is migrated away from the big core.

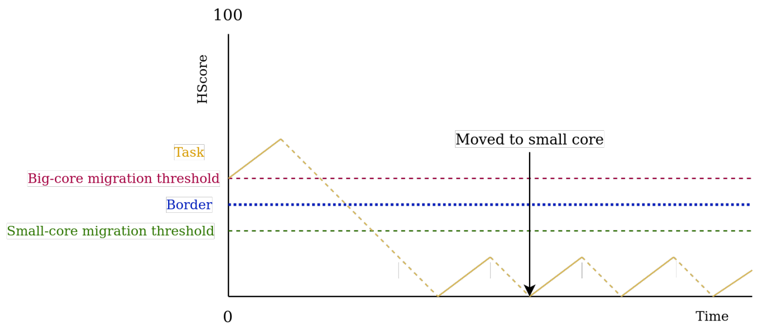


Figure 4.2: Execution of hcs_primes_periodic.

In figure 4.3 we see the execution graph of *hcs_primes*. The difference to *hcs_primes_burst* is that *hcs_primes* waits for user input before starting its calculation. Waiting for user input reduces the HScore, but HCS doesn't move the task to a small core on wakeup (as described in 3.1.2). Instead, the task stays on a big core until the end of its execution, which is the most efficient and performant way to handle it.

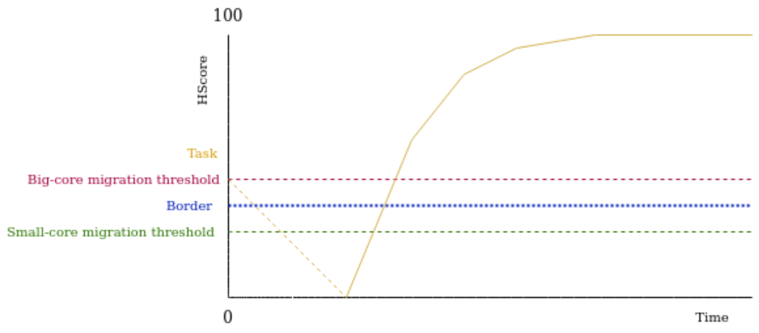


Figure 4.3: Execution of hcs_primes.

Lastly, in figure 4.4 we see the execution graph of *hcs_add3*, an extremely cache-intensive process that does matrix calculations. Even though the process never voluntarily sleeps (high Interactivity Score), we see that its HScore goes below the small-core migration threshold (we don't use the default bias weight of 0.5 for this example, instead opting for a value of 0.8). Nonetheless, the task stays on a big core because active migration doesn't migrate tasks to the small cores. If big cores become highly loaded, this process will be one of the first ones to be moved to a small core.

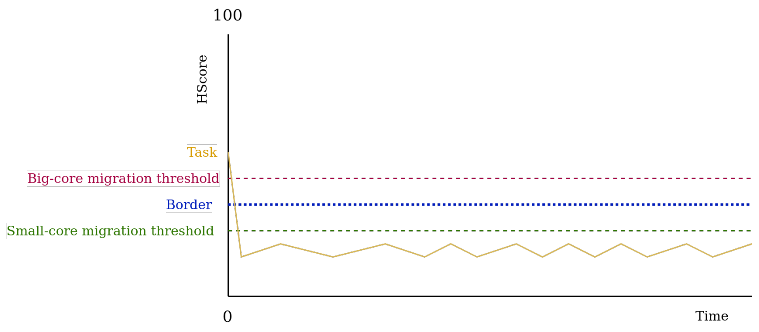


Figure 4.4: Execution of hcs_add3.

An interesting side effect of executing processes that have a high interactivity score (figures 4.1, 4.3, 4.4) is that because ULE passes some of the interactivity score of the child process to the parent process when execution ends, the shell process (which spawns all these processes) is sometimes momentarily moved to a big core. The Quality of Service feature can be used to avoid this behavior.

These validation tests demonstrate HCS's ability to handle a variety of applications and correctly schedule them. All the programs used, and more, are included in the repository of HCS [5].

5. CONCLUSIONS & FUTURE DIRECTIONS

Recently, Asymmetric CPUs have made their appearance in the PC space, bringing energy efficiency and performance improvements. With them, a series of challenges arise for operating system schedulers. Existing heterogeneity-aware schedulers either have obvious deficiencies or are unable to handle Simultaneous Multi-threading (SMT) an important technique widely used in x86 processors.

We proposed HCS, an SMT-aware, Heterogeneity-aware general-purpose scheduler implemented in the Linux Kernel. HCS combines the three main methods of heterogeneous scheduling, utilization-based scheduling, bias scheduling, and proportional-share (fair) scheduling to limit the weaknesses that each method has individually and create a solid general scheduler. HCS is able to handle any system load and can be easily configured to trade throughput for energy efficiency or fairness. HCS is built on top of a ULE port for Linux which we have updated for Linux 5.19.

We presented validation tests that showcase the ability of HCS to handle a number of different tasks. In the future, we are planning to complete the Linux Kernel implementation of HCS, which is missing some relatively minor features, and benchmark HCS on a purpose-built system.

ABBREVIATIONS - ACRONYMS

SMT	Simultaneous Multi-threading
CFS	Completely Fair Scheduler
HCS	Heterogeneous CPU Scheduler
ISA	Instruction Set Architecture
CPU	Central Processing Unit
OS	Operating System
PMU	Performance Monitoring Unit
MIPS	Million Instructions per Second
EAS	Energy Aware Scheduling
CAS	Capacity Aware Scheduling
PELT	Per-Entity Load Tracking
WALT	Window Assisted Load Tracking
IPC	Instructions per Cycle
TLB	Translation Lookaside Buffer
HScore	HCS Score
PSM	Proportional Share Mechanism
QoS	Quality of Service

BIBLIOGRAPHY

[1] Capacity aware scheduling — the linux kernel documentation. `https://docs.kernel.org/scheduler/sched-capacity.html`.

[2] Energy aware scheduling — the linux kernel documentation. `https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html`.

[3] Schedutil — the linux kernel documentation. `https://www.kernel.org/doc/html/latest/scheduler/schedutil.html`.

[4] Optimizing software for x86 hybrid architecture. `https://www.intel.com/content/dam/develop/external/us/en/documents-tps/348851-optimizing-x86-hybrid-cpus.pdf`, 2021.

[5] Github repository of hcs. `https://github.com/SKefalidis/linux-hcs`, 2022.

[6] Apple. Energy efficiency guide for mac apps. `https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/PrioritizeWorkAtTheTaskLevel.html#//apple_ref/doc/uid/TP40013929-CH35-SW1`.

[7] ARM. big.little technology: The future of mobile. `https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf`, 2013.

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[9] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 29–40. Association for Computing Machinery.

[10] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The battle of the schedulers: FreeBSD ULE vs. linux CFS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 85–96, Boston, MA, July 2018. USENIX Association.

[11] Nagabhushan Chitlur, Ganapati Srinivasa, Scott Hahn, P K Gupta, Dheeraj Reddy, David Koufaty, Paul Brett, Abirami Prabhakaran, Li Zhao, Nelson Ijih, Suchit Subhaschandra, Sabina Grover, Xiaowei Jiang, and Ravi Iyer. Quickia: Exploring heterogeneous architectures on real prototypes. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–8, 2012.

[12] hoakley. How macos manages m1 cpu cores. `https://eclecticlight.co/2022/04/25/how-macos-manages-m1-cpu-cores/`, 2022.

[13] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems - EuroSys '10*, page 125. ACM Press.

[14] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[15] Vinivius Petrucci, Orlando Loques, and Daniel Mosse. Lucky scheduling for Energy-Efficient heterogeneous Multi-Core systems. In *2012 Workshop on Power-Aware Computing and Systems (HotPower 12)*, Hollywood, CA, October 2012. USENIX Association.

[16] Jeff Roberson. ULE: A modern scheduler for FreeBSD. `https://papers.freebsd.org/2003/bsdcon/jeff-ule_scheduler/`.

[17] Juan Carlos Saez, Alexandra Fedorova, David Koufaty, and Manuel Prieto. Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems. *ACM Trans. Comput. Syst.*, 30(2), apr 2012.

[18] Juan Carlos Saez, Adrian Pousa, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matias. Towards completely fair scheduling on asymmetric single-isa multicore processors. *Journal of Parallel and Distributed Computing*, 102:115–131, 2017.

[19] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. HASS: a scheduler for heterogeneous multicore systems. 43(2):66–75.