UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

PEDRO HENRIQUE CAPP KOPPER

Topology-Aware Task Scheduling For Heterogeneous Architectures

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Antonio Carlos Schneider
Beck Filho

Porto Alegre
August 2024

“Nothing in life is to be feared, it is only to be understood.
Now is the time to understand more, so that we may fear less.”

— MARIE CURIE

ACKNOWLEDGEMENTS

I believe that this work is the culmination of all the years spent on my undergraduate degree. I would like to first and foremost thank my parents, Carlos and Cecy, for their unconditional support over all those (intense) years. Across pandemics and floods, we stayed strong and were able to achieve our dreams. I would also like to thank Prof. Dr. Antônio Carlos Schneider for his advice and unbreakable patience with every last-minute delivery. Finally, I would also like to thank all my friends who supported me not only during the preparation of this works, but through all the ups and downs of growing not only as as student and professional, but also as a human. The years spent in my undergraduate degree were of intense growth and change, and I am forever thankful to everyone who accompanied me throughout this process.

ABSTRACT

Due to pressing power consumption requirements, recent processors have started to feature heterogeneous, same-ISA cores. This causes modern processors to have highly unique and asymmetric topologies, requiring special attention to task scheduling to obtain high performance and avoid hitting bottlenecks. This work proposes utilizing performance counters to profile applications online and use a machine learning model to map them to the most appropriate cores. We start by characterizing the processor and analyzing the memory subsystem for possible bottlenecks. Once that is identified, performance counter data is collected from several real-world benchmarks. This data is organized as a dataset for training a gradient boosting classifier, which can predict which thread should be scheduled in the most performing core with 91.1% accuracy. This model is then used to develop a thread placing script, which dynamically sets the CPU affinity of threads based on the model. This strategy can lead to up to 69.9% performance gain on select benchmarks, with a geometric average of 7.76%. Finally, regressing benchmarks are analyzed to improve the model in the future.

Keywords: Heterogeneous architectures. cache. scheduling.

Escalonamento de Tarefas Ciente de Topologia para Arquiteturas Heterogêneas

RESUMO

Devido aos requisitos atuais de consumo de energia, os processadores mais recentes começaram a apresentar núcleos heterogêneos e que utilizam a mesma ISA. Essa característica faz com que os processadores modernos possuam topologias altamente exclusivas e assimétricas, exigindo atenção especial no escalonamento de tarefas para obter alto desempenho e evitar gargalos. Começamos caracterizando o processador e analisando o subsistema de memória para encontrar possíveis gargalos. Uma vez que isso é identificado, os dados de *performance counters* são coletados de vários *benchmarks* de mundo real. Esses dados são organizados como um conjunto de dados para treinar um classificador de aumento de gradiente, o qual pode prever qual *thread* deve ser alocada no núcleo de melhor desempenho com 91,1% de precisão. Esse modelo é então usado para desenvolver um script de alocação de cores, que define dinamicamente a afinidade de CPU das threads com base no modelo. Essa estratégia pode levar a um ganho de desempenho de até 69,9% em benchmarks selecionados, com uma média geométrica de 7,76%. Finalmente, os benchmarks que regrediram são analisados para melhorar o modelo no futuro.

Palavras-chave: Arquiteturas Heterogêneas, Escalonamento.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
CFS	Completely Fair Scheduler
CPU	Central Processing Unit
EEVDF	Earliest Eligible Virtual Deadline First
ISA	Instruction Set Architecture
PMC	Performance Monitoring Counters
SF	Speedup Factor
SMP	Symmetric Multi-Processor
SMT	Simultaneous multithreading
TMA	Top-down Microarchitecture Analysis
TSC	Time Stamp Counter

LIST OF FIGURES

LIST OF TABLES

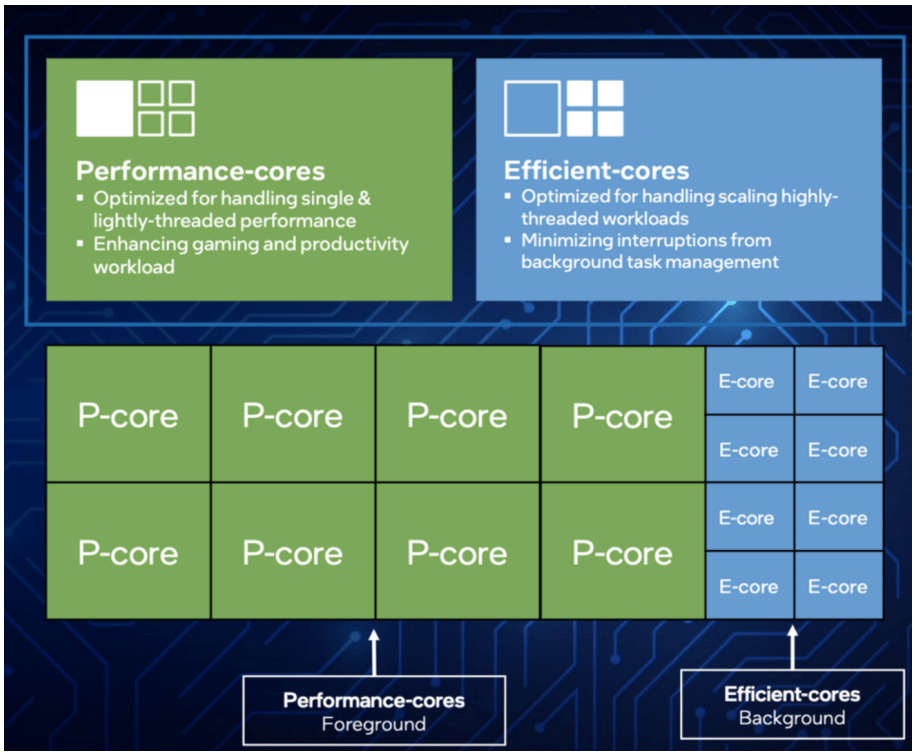
CONTENTS

1 INTRODUCTION

Recent demands in mobile computing have pressured the industry to drive down power consumption rapidly, aiming to increase battery life and reduce heat dissipation. This has led to significant changes in computer architectures, most notably the rise in heterogeneous architectures, where more than one micro-architectures of the same ISA are available in the same package (KUMAR et al., 2003).

One of the first of such architectures was Arm's big.LITTLE, introduced with the Cortex-A7 and Cortex-A15 pair (ARM, 2011). While both cores are compatible, the A7 is a simpler in-order core, while the A15 features a faster, but significantly more complex design, featuring an out-of-order pipeline capable of speculative execution. This concept was further expanded with the introduction of the *DynamIQ* technology, allowing for even more core cluster configurations, while also allowing them to scale independently in frequency and voltage and providing shared and coherent memory access (ARM, 2017).

Figure 1.1: Intel Alder Lake Heterogeneous Architecture Announcement



Source: Adapted from (Intel Corp., 2021)

The previously cited technologies applied mostly to smartphones, tablets, and other mobile devices. However, with the introduction of Apple's M1 processor in 2020 (Apple Inc, 2020), based on an Arm ISA, there was a surge in interest in heterogeneous architectures on desktop and mobile personal computers. Intel quickly followed suit, re-

leasing the x86-64 Alder Lake processors, featuring a mixture of Golden Cove and Gracemont cores (Intel Corp., 2021) with the proposed architecture in Figure 1.1. Each core was optimized for performance (P-cores) or area efficiency (E-cores), respectively, while both featured out-of-order superscalar pipelines. Recent designs, such as Intel's Meteor Lake (Intel Corp.,) go a step further, featuring an even lower tier called LP-cores. These cores are located on a separate die and can be powered on and off independently from the rest of the chip, allowing for even higher power savings.

To use these processors efficiently, while still keeping a satisfactory throughput, the operating system must intelligently schedule tasks across cores, taking into account their performance, power consumption, and position in relation to its siblings. Also, it might be able to turn off certain cores entirely, achieving high energy savings, but having to handle an additional wake-up latency. Therefore, considerable strain is put into the scheduler and various approaches have been developed to achieve these tasks quickly and effectively, ranging from hardware feedback to user-space daemons.

One possible route for optimization is exploiting the memory hierarchy differences to provide better performance. In some special cases, the performance improvement compared to naive thread placement can be up to 36% (ZHANG; JIANG; SHEN, 2010). Some effort has also been made to determine an analytical model of the memory hierarchy of multi-core processors (MOHAMED; MUBARK; ZAGLOUL, 2023), demonstrating the performance impact of the variance in memory access time due to interdependence between different memory layers. Other authors also suggest that cache-aware scheduling could be beneficial for achieving higher energy savings, although this will not be explored explicitly in this thesis (SHEIKH; PASHA, 2022).

In the past, there have been attempts to achieve better performance by improving data locality using compilation-time techniques (ZHANG; KANDEMIR; YEMLIHA, 2011) (JIANG et al., 2011) (KANDEMIR et al., 2010). In these cases, the multi-layer cache hierarchy is characterized by a *reuse distance* metric, which is tied to the latency from each core to each cache. However, this requires *a priori* knowledge of the program under optimization, limiting its applications.

On the other hand, when considering raw performance differences across cores we can optimize for different targets. For instance, on mobile platforms, authors tend to optimize for energy efficiency, as it is usual for lower-performance cores to be more energy-efficient. At the same time, when dealing with area-efficient cores, the energy penalty is not as significant as the performance one, so maximizing throughput using all

available cores is a more interesting goal.

In this work, we will be exploring how we can optimize the scheduling of tasks for the Linux kernel on an Intel Alder Lake microprocessor. We will do this by first exploring the potential memory differences to identify whether there is any gain to be obtained there. Further, we will explore the performance gap between the two tiers of cores available on our microprocessor by running real-world workloads and measuring performance and hardware counters. Using that data, we will then build a machine learning model that can estimate which type of core a thread should be placed in, aiming to optimize the case of multi-task systems where multiple programs compete for limited CPU resources. By leaving performance cores only to the threads that will benefit the most and moving the rest to the efficiency cores, we can achieve less throughput degradation. Finally, we will validate our model by showcasing a CPU-affinity-based scheduling overlay, which will run on top of a stock Linux image and be used for performance evaluation.

Overall, the main contributions of this thesis are:

1. Micro-benchmarks of the memory access differences between Golden Cove and Gracemont cores
2. Review of the performance differences between Golden Cove and Gracemont cores using real-world workloads
3. Development of a dataset of different applications and their performance counters on Alder Lake processors
4. Development of a machine learning classifier for thread placement on Alder Lake processors
5. Implementation of a scheduler-overlay using a machine model that optimizes thread-to-core mapping on Alder Lake processors

1.1 Organization

This thesis's main contributions are centered around modern x86 heterogeneous architectures for consumer use. Starting in Section 2, we review the micro-architecture of the Intel i7-1260P mobile processor. Then, we review how the Linux kernel currently handles asymmetric topologies, both in scheduling and power management decisions. In Section 3, we then perform a series of benchmarks based on characteristics identified during the bibliography review, aiming to better understand how they affect performance

and energy consumption. We start by investigating the difference in cache access latency and measuring its impact on memory-bound applications. After discarding this hypothesis, we follow up by performing a higher-level investigation comprising a series of application-level benchmarks monitoring hardware performance counters. Finally, we present a machine-learning model built using this data that can pick the most suitable core for each task based on its performance counters.

2 BIBLIOGRAPHICAL REVIEW

In this section, we will start by giving a basis for the work being developed. First, fundamentals of computer architecture will be covered, as well as a review of the processor under analysis. Following that, we will review how the Linux scheduling subsystem works as to provide a basis for the development of our own scheduler, reviewing how it works in general and how it currently handles heterogeneous architectures. Then, we will cover machine learning basics, as this will be a core mechanism for our thread-classifying algorithm. Finally, we present a summary of the related work done in this area.

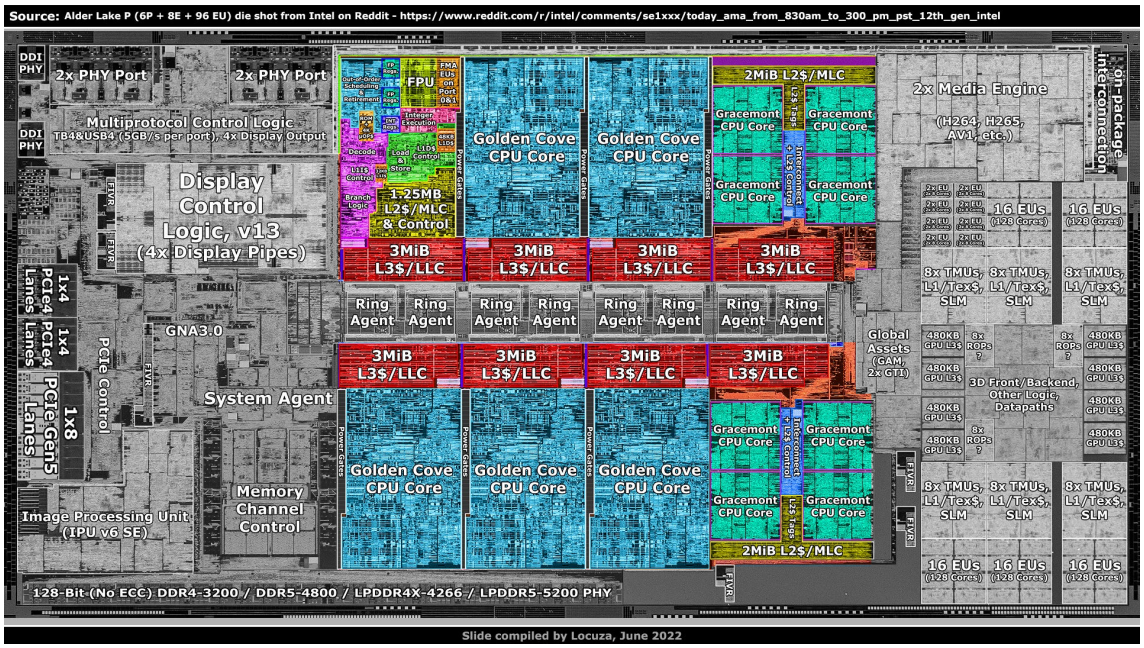
2.1 Heterogeneous Architectures

Traditionally, multi-core systems have been composed by having multiple cores using the same micro-architecture. However, as the need for more energy-efficient designs appeared, it was proposed by (KUMAR et al., 2003) to evaluate single-ISA heterogeneous multi-core architectures as a mechanism to reduce processor power dissipation. By having multiple core types available, each type could be optimized for a specific purpose, e.g. high performance or area/power efficiency. At the same time, retaining the same ISA across cores allowed software unaware of the heterogeneity to keep running without modifications, easing adoption. In the cited system, the authors were able to reduce 39% on average energy while running 14 SPEC benchmarks, while sacrificing performance by only 3%. This left the question of not only how to size these cores, but how to schedule threads efficiently between them, especially since load characteristics can vary during program execution. For instance, programs can spend only a small portion of their runtime running intense computations while spending a lot of time waiting on I/O.

By implementing this technique, Intel can fit more processor cores into a single die. For instance, in Figure 2.1 we can see the Golden Cove (performance) cores in blue on the left, while Gracemont (efficiency) cores are in green on the right. Using the same area of a Golden Cove core, there can be four Gracemont cores in a cluster. This means a processor in the example can have $6 + 8 = 14$ cores instead of 8 cores if they were all performance-focused. This way, a single processor can achieve a high core count, useful for highly parallel applications, while retaining high performance on single-threaded ones.

When analyzing processors from the Alder Lake generation running integer bench-

Figure 2.1: Intel Alder Lake Annotated Die Shot



Source: Adapted from (LOCUZA, 2022)

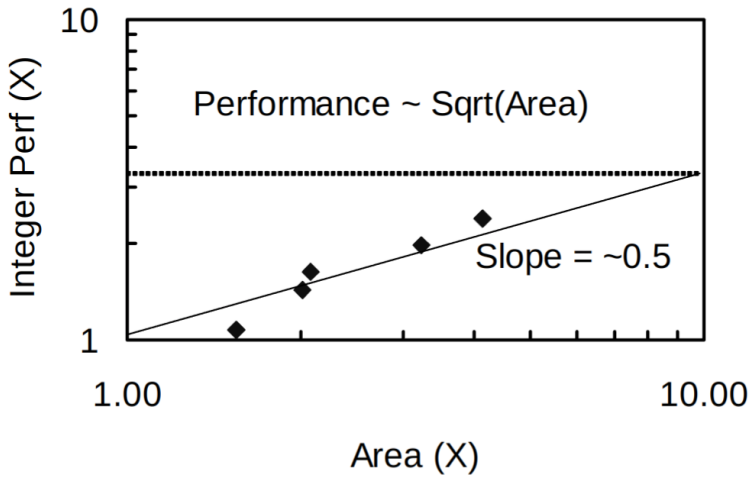
marks, the geometric mean performance of an efficiency core is $0.8\times$ that of a performance core running at the same frequency, while for floating point it can reach $0.62\times$ (ROTEM et al., 2022). This is in line with Pollack's rule, which states that the performance of a processor increases proportionally to the square root of complexity as seen in Figure 2.2 (BORKAR, 2007). Therefore, a processor twice as large as the other should only provide a roughly 41% increase in performance, despite a much larger area. This difference can vary per program and depends on the instruction mix and other factors, which will be estimated during the course of this thesis. These distinctions will form the basis of our model.

2.2 Intel Core i7-1260P Topology

For this thesis, the analysis will be focused on the Intel Core i7-1260P processor, due to its availability to the author. According to the data on its product page (Intel Corp., 2022), the Intel Core i7-1260P is a mobile Alder Lake processor manufactured using the Intel 7 lithography. It provides 12 cores, of which 4 are optimized for performance and 8 for efficiency. Considering that only the performance cores support simultaneous multi-threading, this allows the processor to execute up to 16 threads simultaneously.

Its performance cores are based on the Golden Cove micro-architecture and can

Figure 2.2: Pollack's Law For Microprocessors



Source: (BORKAR, 2007)

run up to 4.70 GHz, depending on thermal constraints. At the same time, efficiency cores are based on the Gracemont micro-architecture and can run up to 3.40 GHz. Looking deeper into its datasheet (Intel Corp., 2023a), we can understand the cache topology used in Alder Lake processors. In the P-cores, the first level cache is divided into a data and an instruction cache, providing 48KB and 32KB respectively. Both of the caches are 12-way associative. In the E-cores, the first level cache is divided into a data and an instruction cache. Still, they are sized differently, providing 32KB for data and 64KB for instructions, both being 8-way associative.

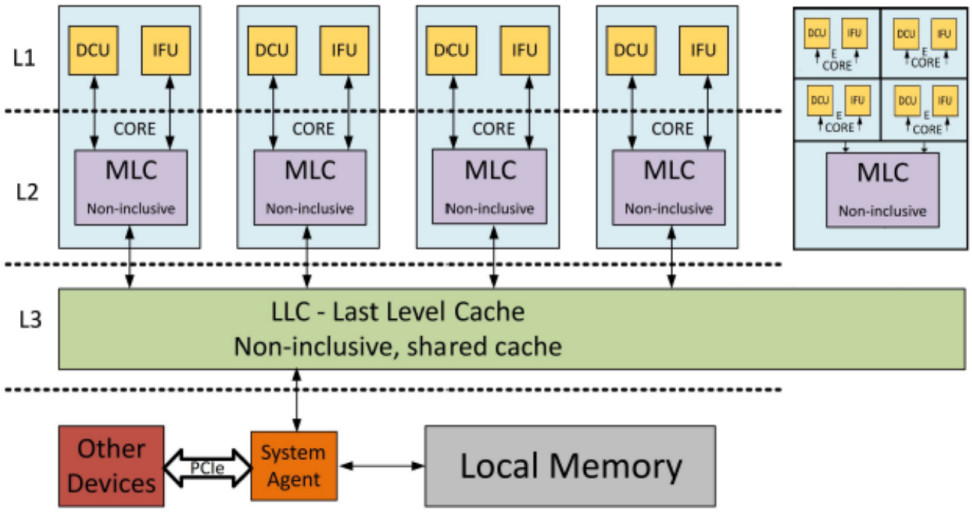
The second level is private for P-cores, providing 1.25MB of a 10-way non-inclusive associative cache. However, in the E-cores it is shared in clusters of four cores, providing 2MB of 16-way non-inclusive associative cache for each cluster.

The third and last level is shared among all cores and processor graphics core. Its size varies depending on the product number, being 18MB in the case of the i7-1260P. It is 12-way non-inclusive associative.

This data is exposed in a machine-readable format to the operating system. In Linux, this is accessible to userspace through a *sysfs* interface under */sys/devices/system/cpu/*. There are tools, such as *hwloc*, that allow parsing this data and generating graphics, which as the one in Figure 2.4.

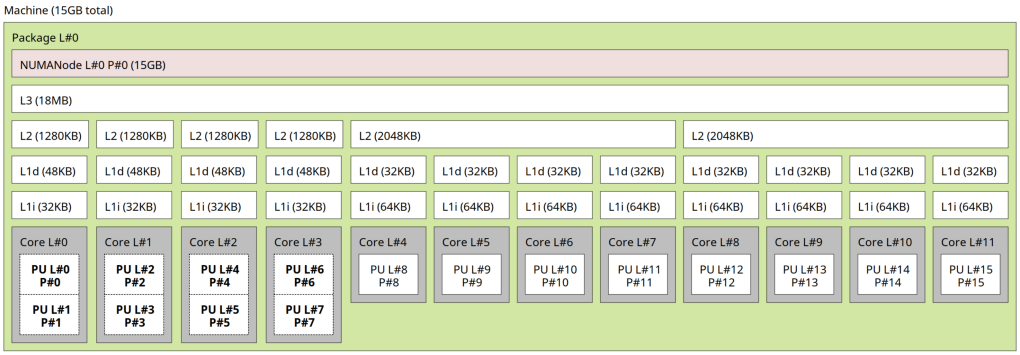
Processor numbering follows the Linux convention, where SMT pairs are grouped. Thus, P#{0-7} are P-cores, where P#{1,3,5,7} are the SMT siblings of P#{0,2,4,6}. Processors P#{8-15} are E-cores. It is important to note how each P-core has its private L2 cache, while E-cores share two L2 caches in clusters of 4. This leads to a difference in

Figure 2.3: Intel Hybrid Cache Architecture



Source: (Intel Corp., 2023a)

Figure 2.4: i7-1260P memory architecture



Source: The author

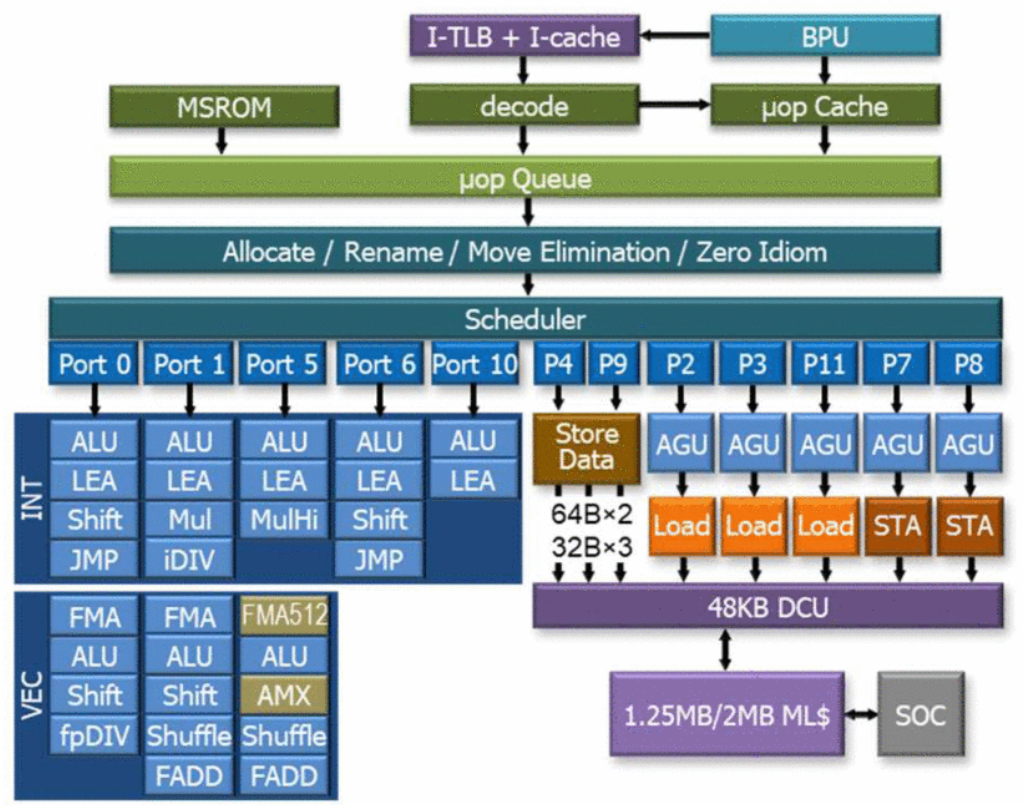
inter-core memory latency that will be subsequently explored. We will now present a deeper analysis of each core type.

2.2.1 Golden Cove (P-cores)

The Golden Cove micro-architecture is used both in Alder Lake for consumer devices and in Sapphire Rapids for servers. It is focused on having a wide out-of-order core, with the infrastructure around it to keep it fed. Its frontend has a decode pipeline fetch bandwidth of 32 bytes/cycle, used to feed its six decoders, making it capable of decoding up to six instructions per cycle. Accordingly, the backend has a 6-wide rename/allocation unit, paired with 12 execution ports. Finally, instructions can be retired out-of-order from a 512-entry window. They can also support simultaneous multi-threading (SMT)

and support extra vector extensions, such as AVX-512, that lead to higher performance but will not be explored in this thesis (ROTEM et al., 2022). Due to all this, the core is significantly larger in area and consumes more power, however, it can more effectively achieve higher parallelism and, consequently, throughput.

Figure 2.5: P-core block diagram

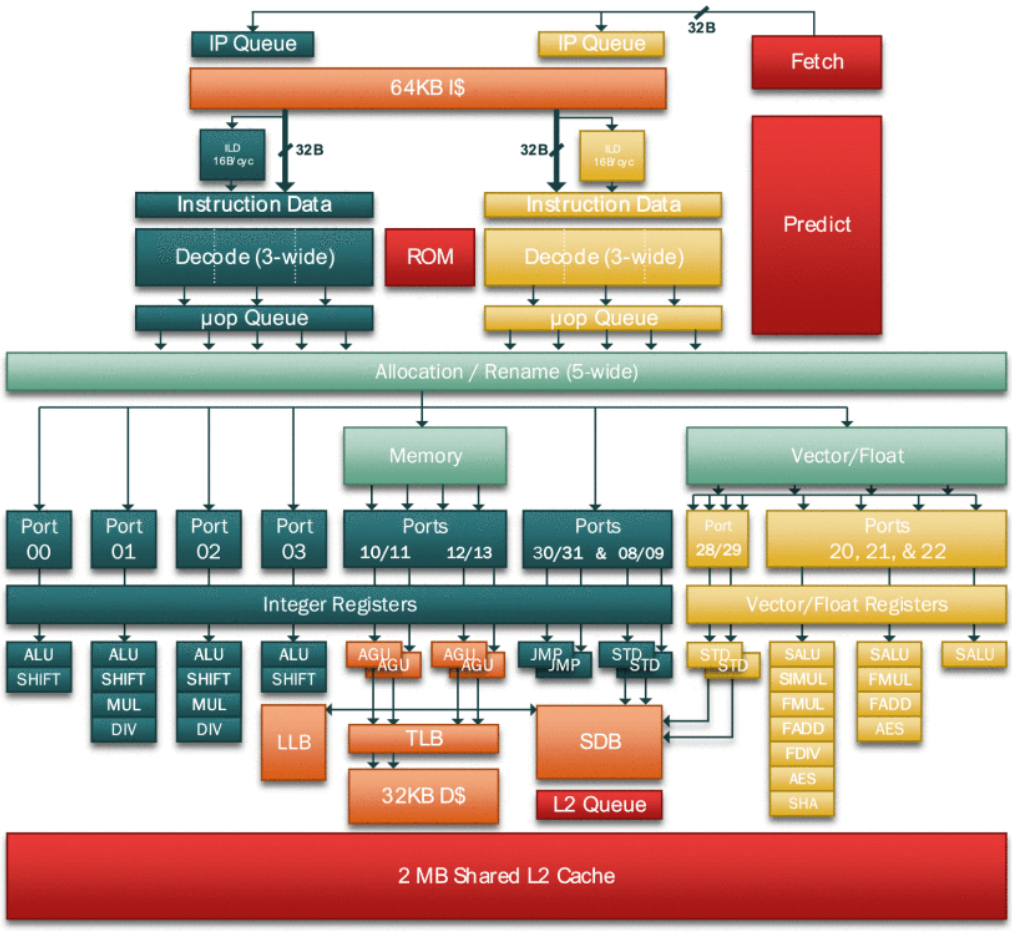


Source: (ROTEM et al., 2022)

2.2.2 Gracemont (E-cores)

At the same time, Gracemont cores have different design goals, being optimized in terms of area and power. We have a five-issue core with six decoders, which are grouped in two clusters. Each cluster takes data from an instruction pointer queue, which is delivered in-order at 32 bytes/cycle. Every taken branch then toggles between clusters. In the backend, there is a five-instruction-wide allocation unit, which reads the micro-operation queue from both clusters in-order. Finally, up to eight instructions can be retired out-of-order with a 256-entries window. (ROTEM et al., 2022). Combined with a smaller cache, this design leads to a significantly smaller design, albeit slower performing as explained in Section 2.1.

Figure 2.6: E-core block diagram



Source: (ROTEM et al., 2022)

2.3 The Linux Scheduling Subsystem

In an operating system, the job of multiplexing the CPU across multiple threads is performed by the scheduler. Its objective is to allow multiple competing tasks to share the same processors fairly while maximizing throughput and minimizing latency. As we will be overriding part of the scheduler behavior using CPU affinity, it is important to first understand how it works and its purpose. In this section, two Linux schedulers will be described.

2.3.1 Completely Fair Scheduler (CFS)

Since Linux 2.6.23, the default scheduler has been the Completely Fair Scheduler (CFS) (TORVALDS, 2024, 6.8). It aims to model an ideal, precise multi-tasking CPU, which could theoretically run all tasks at equal speed in parallel. On real hardware, however, it divides the CPU using the "virtual runtime" concept, modeling when the next time slice would start executing on an ideal CPU, normalized by the total number of running tasks.

Ideally, all tasks would have the same virtual runtime. In practice, that does not hold, so CFS builds a time-ordered red-black tree that models a "timeline" of future task execution, sorted by virtual runtime. It then runs the leftmost (i.e. the least ran so far) until a scheduler tick happens. Then it increases the virtual runtime, rebalances the tree and, if another task becomes the leftmost, the current task is preempted and swapped. Finally, when adding a new task it gets assigned a configurable minimum amount of virtual runtime.

2.3.2 Earliest Eligible Virtual Deadline First (EEVDF)

Starting on the Linux Kernel version 6.6, CFS has been replaced by the Earliest Eligible Virtual Deadline First (EEVDF) scheduler. First proposed in 1995 (STOICA; ABDEL-WAHAB, 1995), it builds on the *virtual time* concept to track the work done by each task. However, it also introduces the idea of a *virtual deadline* and *lag*. After running a task for a given time, the scheduler updates the *lag* to be equal to the time the task actually runs minus its allotted time. A task only becomes *eligible* when its *lag* is

positive (i.e. it is "owed" CPU time). To calculate the *virtual deadline*, it computes the time remaining in its time slice to the time it became eligible. Then, the *eligible* task with the earliest *virtual deadline* is run. This ensures that not only does each task get a fair share of CPU time, but also that tasks with shorter time slices (presumed to be interactive or latency sensitive) will tend to run first, improving the system responsiveness.

2.4 Current scheduling approaches and optimizations

Currently, the Linux Kernel documentation (TORVALDS, 2024, Version 6.8) specifies two core strategies for dealing with heterogeneous architectures from a scheduler standpoint. Two models run concurrently, a capacity and an energy one, estimating processing power and energy consumption respectively.

2.4.1 Capacity Aware Scheduling

The Linux Kernel utilizes internally a metric known as *capacity* to model differences between CPUs in heterogeneous architectures. This is defined as a measure of a given CPU's performance, normalized against the most performing CPU in the system. This metric is then internally made frequency-invariant by dividing it by the frequency of the highest operating point for each CPU. Finally, all task utilization metrics are also normalized by frequency. These metrics are then fed into the Completely Fair Scheduler (CFS) to allocate tasks to CPUs while ensuring they fit into the CPU's capacity and are balanced. However, this mechanism relies on capacity measurements provided by the manufacturer and doesn't consider other types of micro-architectural differences between CPUs, tying everything up into a single scalar value.

2.4.2 Energy Aware Scheduling

The Energy Aware Scheduling (EAS) subsystem allows the kernel to estimate the impact of its decisions on the energy consumption by the CPUs, using a given Energy Model. This model is provided using a dedicated framework and maps given performance levels to power consumption using either a mathematical approximation or feedback from the hardware of the device. Using this information, it is possible to use the duration of

a task to estimate the total energy consumed by running it on a given core. Compared to CFS alone, it extends it by prioritizing CPUs with the highest spare capacity in each performance domain, which allows the frequency to be kept to a minimum. Finally, it checks whether placing the task there would save energy compared to leaving it at the previous CPU. If that is the case, the task is then moved. It is important to note that this behavior is intrinsically tied to the accuracy of the Energy Model, as all decisions are made on top of its data.

2.4.3 Intel Thread Director

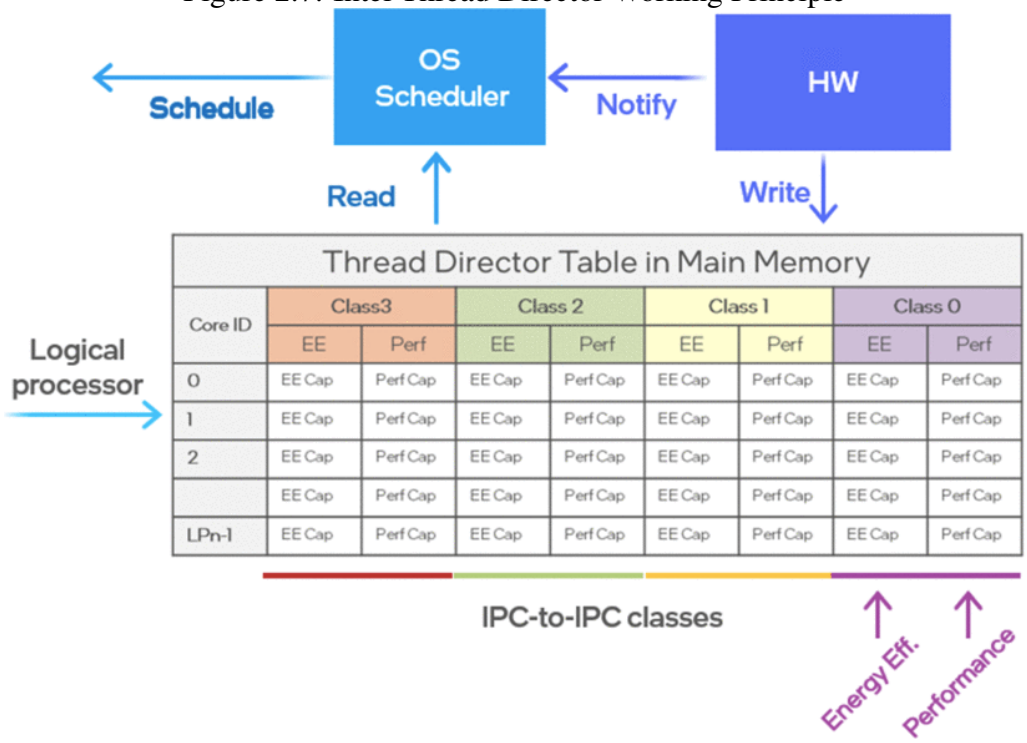
The Intel Thread Director technology is a hardware subsystem that was first introduced in the Alder Lake to help out with operating system scheduling decisions (Intel Corp., 2023b). It is implemented alongside the Hardware Feedback Interface (HFI) as a table in memory (described in Figure 2.7), which provides per-thread guidance to the OS based on a proprietary algorithm. At the same time, HFI provides real-time information on per-core performance and energy efficiency capabilities. Using these two data sources together should provide a complete solution for choosing the appropriate CPU core per thread.

However, although it is described as "optimal" in the official documentation, there seems to be evidence against this case. In a study focusing on the Intel Core i9-12900K (SAEZ; PRIETO-MATIAS, 2022), it was found that the Thread Director only predicts a fixed speed-up factor for each of its four classes. This, combined with the fact that 99.9% of the Thread Director readings were classified only as Class 0 or Class 1, suggests that Intel's Thread Director might lack the granularity needed for fine-grained speed-up estimation, leading to less-than-optimal scheduling solutions.

2.4.4 Collaborative Processor Performance Control

The Collaborative Processor Performance Control (CPPC) is a mechanism defined in the APCI specification that allows the operating system to understand the performance of logical processors on a continuous and abstract performance scale (TORVALDS, 2024). It exposes multiple registers for each CPU via a *sysfs* interface as described in Table 2.1.

Figure 2.7: Intel Thread Director Working Principle



Source: (ROTEM et al., 2022)

When measuring these values on the Intel i7-1260P, the values in Table 2.2 are returned. This indicates that the performance difference between P-cores and E-cores must be $1.76x$ at peak performance and $1.24x$ sustained, which is in line with what the bibliography described earlier.

2.5 Machine learning

As we will be dealing with multi-modal data classification, we decided to pursue a machine learning (ML) approach to the problem. It uses statistical models to enable computers to learn from data, make decisions, and improve their performance on a specific task without being explicitly programmed. Machine learning is a suitable approach for multi-modal data classification because it can effectively handle diverse data types, such as different performance counters, by combining features from each modality into a single representation. This allows the model to leverage the strengths of each input type, improving overall performance and robustness. Additionally, machine learning algorithms can automatically adapt to changes in the data distribution or new modalities, making them more scalable and maintainable than traditional rule-based approaches (ALPAYDIN, 2014).

Table 2.1: CPPC *sysfs* interface.

sysfs entry	**Description**
highest_perf	Highest performance of this processor (abstract scale).
nominal_perf	Highest sustained performance of this processor (abstract scale).
lowest_nonlinear_perf	Lowest performance of this processor with nonlinear power savings (abstract scale).
lowest_perf	Lowest performance of this processor (abstract scale).
lowest_freq	CPU frequency corresponding to lowest_perf (in MHz).
nominal_freq	CPU frequency corresponding to nominal_perf (in MHz). The above frequencies should only be used to report processor performance in frequency instead of abstract scale. These values should not be used for any functional decisions.
feedback_ctrs	Includes both Reference and delivered performance counter. The reference counter ticks up proportional to the processor's reference performance. The delivered counter ticks up proportional to the processor's delivered performance.
wraparound_time	Minimum time for the feedback counters to wraparound (seconds).
reference_perf	Performance level at which reference performance counter accumulates (abstract scale).

Adapted from (TORVALDS, 2024)

Out of the several possible models, including neural networks and their variants, gradient boosting was selected as the method of choice for building the classifier in this work.

2.5.1 Gradient boosting

When building either regression or classification models from tabular data, a common approach is using gradient boosting to build the model, first proposed by (FRIEDMAN, 2001). They allow for models to be built without having a previously specified, expert-written model of the data. Instead of the traditional approach of building a single, strong model, it trains an ensemble of weak learning models, which are added sequentially to minimize the loss function (NATEKIN; KNOLL, 2013). Each new model is trained to maximize its correlation with the negative gradient of the loss function. This process is repeated until convergence, resulting in a final model that can be highly accurate and robust. The choice of loss function is flexible, allowing gradient boosting to be tailored to

Table 2.2: CPPC values on the Intel i7-1260P

Register	**P-core**	**E-core**
feedback_ctrs	ref:40491209600 del:41968755597	ref:19009428200 del:11712204445
highest_perf	60	34
lowest_freq	0	0
lowest_nonlinear_perf	12	10
lowest_perf	1	1
nominal_freq	2100	2100
nominal_perf	26	21
reference_perf	31	25
wraparound_time	18446744073709551615	18446744073709551615

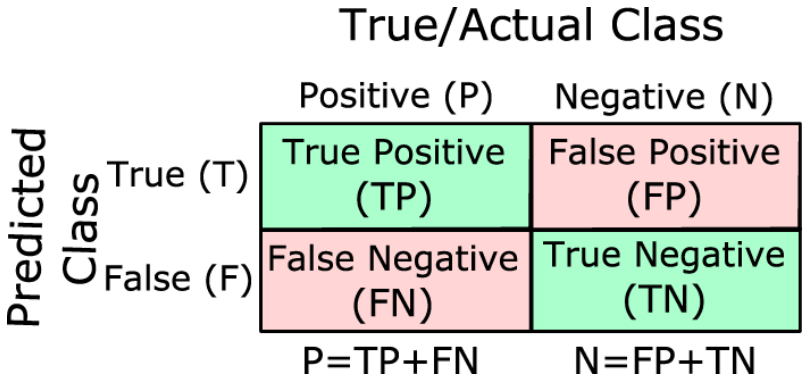
Source: The author

specific data-driven tasks. This technique has shown success in various machine learning and data mining challenges and is particularly useful for predictive tasks involving sensor data. Therefore, it will be used to analyze our data further down the line.

2.5.2 Classifier performance evaluation

After building a model, it is important to have metrics that evaluate how accurate its predictions are. For this, we will be using recall, precision, and f-scores as defined by (THARWAT, 2020). First, we define the concept of a confusion matrix, which is represented in Figure 2.8. It is a 2×2 matrix for binary classification, where the green diagonal represents correct predictions and the pink diagonal indicates incorrect predictions. We define true positives (TP) and true negatives (TN) as being the cases where the samples are correctly identified as being in the positive or negative class, respectively. If a positive sample is mispredicted to be negative, it is classified as a false negative (FN) and if the opposite happens it is a false positive (FP).

Figure 2.8: Confusion Matrix Definition



Source: (THARWAT, 2020)

Using the definitions above, we can start defining our metrics. First, we define accuracy as being:

$$Acc = \frac{TP + TN}{TP + TN + FP + FN}$$

We can also define precision as the positive prediction value, defined as

$$Precision = \frac{TP}{FP + TP}$$

It represents the proportion of positive samples that were correctly classified to the total number of positive predicted samples. Recall, also called sensitivity or hit rate, is defined as $TPR = \frac{TP}{TP+FN}$. Finally, we can define $F_1\text{-}score$ as the harmonic mean of precision and recall as just defined, resulting in the following expression:

$$F_1 = \frac{2TP}{2TP + FP + FN}$$

It ranges from zero to one and high measures indicate higher classification performance (THARWAT, 2020). Down the line, these metrics will allow us to evaluate the performance of our model for thread placement.

2.6 Related work

In this section, we will first analyze how the memory subsystem impacts the performance of multi-threaded workloads, as this will be one of the characteristics analyzed by our scheduler. After that, we provide an overview of previous attempts at building schedulers optimized for heterogeneous architectures, focusing on those using performance counters.

2.6.1 Cache sharing impact on multi-threaded workloads

The memory system plays a crucial role in the functioning of a CPU, as it provides the necessary storage and retrieval mechanisms for data processing. A CPU relies heavily on its memory hierarchy, comprising cache, main memory, and storage devices, to store and access program instructions and data. A robust memory system enables efficient data transfer between different levels of memory, reducing latency and increasing overall sys-

tem performance. However, it is also a fundamental performance and energy bottleneck in almost all computing systems (MUTLU; MEZA; SUBRAMANIAN, 2015). While CPU technology has been scaling very quickly, DRAM (Dynamic Random-Access Memory) technology has been advancing at a much slower pace. This has been mitigated by adding more cache levels closer to the CPU. By trading off smaller sizes for decreased latency, they can provide a speed-up by storing commonly used data near the CPU. However, this comes at the cost of increased complexity and fragmentation, as all these layers should be transparent to the end-user.

On heterogeneous CPUs, it is important to note that the cache structure is not homogeneous either, both in size and latency. This requires special attention, as the operating system scheduler must be aware of these characteristics to make efficient placement decisions. In some special cases, the performance improvement compared to naive thread placement can be up to 36% (ZHANG; JIANG; SHEN, 2010). Some effort has also been made to determine an analytical model of the memory hierarchy of multi-core processors (MOHAMED; MUBARK; ZAGLOUL, 2023), demonstrating the performance impact of the variance in memory access time due to interdependence between different memory layers. Other authors also suggest that cache-aware scheduling could be beneficial for achieving higher energy savings, although this will not be explored explicitly in this paper (SHEIKH; PASHA, 2022).

In the past, there have been attempts to achieve better performance by improving data locality using compilation-time techniques (ZHANG; KANDEMIR; YEMLIHA, 2011) (JIANG et al., 2011) (KANDEMIR et al., 2010). In these cases, the multi-layer cache hierarchy is characterized by a *reuse distance* metric, which is tied to the latency from each core to each cache. However, this requires *a priori* knowledge of the program under optimization, limiting its applications. Finally, there have also been studies showing the importance of reducing contention on critical shared resources, such as the memory controller and on-chip networks, by appropriately mapping applications to cores (DAS et al., 2013).

In this thesis, we will start by analyzing the impact of the cache directly by utilizing micro-benchmarks, aiming to better understand the role that memory plays in the performance differences among performance and efficiency cores.

2.6.2 Performance counter-based schedulers

In the academy, various research groups have come up with different proposals to model the performance differences between cores better and schedule tasks on heterogeneous processors. For instance, a joint research group between Intel, MIT, and Ghent University has proposed using a Performance Impact Estimation (PIE) metric (CRAEYNEST et al., 2012), which consists of an aggregate measurement of cycles per instruction of both memory and non-memory-related components, and instruction and memory level parallelism. This is based on the assumption that small cores are inherently in-order, while only the big cores are out-of-order. This is not true in modern systems, such as the Gracemont cores analyzed in this thesis. Still, the Gracemont cores have significantly smaller structures to aid in exploring instruction/memory level parallelism, making this a plausible heuristic for determining which core is better suited to each task.

Some systems have already been put in place that allow per-thread performance counter measurement, such as PMCTrack (SAEZ et al., 2017). This module exposes the counters inside the kernel to the scheduler so that it can use performance data on its decisions, while also storing data for offline analysis. The same group has since then built PMCsched, which is a modular scheduling subsystem for Intel Alder Lake platforms using PMCTrack facilities to improve load distribution across asymmetric cores (BILBAO; SAEZ; PRIETO-MATIAS, 2023). During their benchmarking, which comprised running pairs of single-threaded programs derived from SPECcpu, they were able to improve up to 30% throughput gain when compared to the naive Linux scheduler and up to 22% when compared to Intel Thread Director.

Finally, there are classifier-based approaches that utilize machine learning techniques to estimate the best thread-to-core mapping (BORAN; YADAV; IYER, 2020). They utilize online hardware performance counters, similar to the previously cited approach, and break down each benchmark into several phases, classifying each of them individually. The counter's data is fed to a Linear Regression Neural Network (LRNN), which outputs a binary classification. With this approach, they claim to obtain an average speedup of 35.7% with respect to a baseline single ISA heterogeneous architecture.

3 DATA COLLECTION

Given the heterogeneous nature of the processor analyzed in the previous section, combined with the asymmetric cache organization, we hypothesized that we could improve task scheduling compared to the Linux 6.8 scheduler. To evaluate this, we will first characterize the processor, aiming to identify any existing bottlenecks or other characteristics that might be relevant for scheduling. We will do so by performing micro-benchmarks that stress the memory subsystem and measure relevant performance counters.

Next, we will perform benchmarks consisting of real-world applications on both performance and efficiency cores, measuring performance counters to build a dataset. Finally, we will analyze this dataset with machine learning techniques to build a model capable of choosing which of two threads is the most CPU-bound one. Finally, we will apply this model in a script that overrides part of the existing EEVDF scheduler, experimenting with how CPU cores are assigned to tasks aiming to achieve higher throughput when multiple tasks are competing for CPU resources.

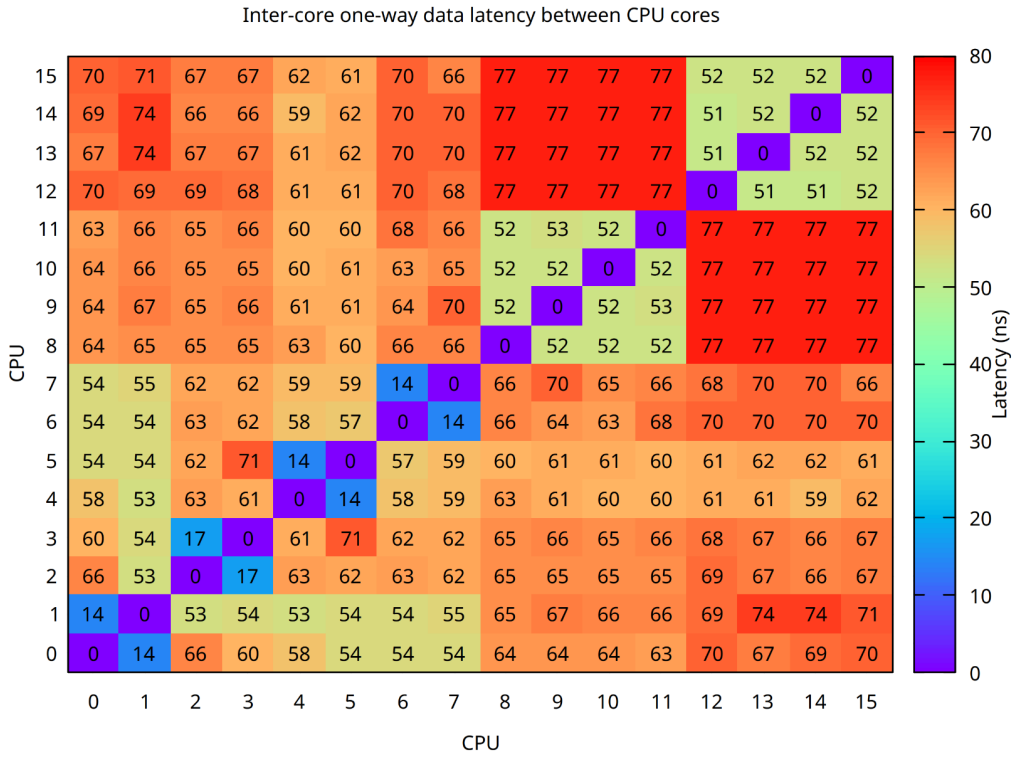
3.1 Processor Characterization

To be able to analyze the impact of the CPU topology on scheduling, it is interesting to first gather data regarding the core-to-core communication latency, as this might provide a useful metric to predict the impact of allocating similar threads to sibling cores. We are interested in collecting this data as a heuristic to identify possible bottlenecks to be explored during the development of our scheduler.

The data was collected using the *c2clat* tool, made available by (RIGTORP, 2020). It measures the latency by spawning two threads, each pinned to a different core. They alternate in locking a mutex, measuring the round-trip time to propagate the lock. The core numbering follows the Linux convention, where SMT pairs are grouped together. Thus, cores 0-7 are P-cores, where 1,3,5,7 are the SMT siblings of 0,2,4,6. Cores 8-15 are E-cores, with 8-11 and 12-15 being on separate clusters.

In Figure 3.1 above, we can identify the same-core latency of 0 in the diagonal. SMT siblings, which are only available in the P-cores, show the lowest latency between 14 and 17ns. Of course, there is no true cross-core communication in this case, as they share the same physical core. The lowest true core-to-core latency, however, shows up in the top right quadrant. While communicating across E-core clusters causes the highest

Figure 3.1: Core-to-core latency on a i7-1260P



Source: The author

latency possible, around 77ns, the lowest one is achieved within E-cores on the same cluster, dropping to between 51 and 52ns. This up to 51% difference in latency can be significant, both for improving our scheduler or decreasing the performance of unaware ones. The impact of this difference in latency will be analyzed in the following sections.

3.2 Cache versus performance

While the Core i7 1260P performance cores are based on the Golden Cove micro-architecture and can run up to 4.70 GHz, depending on thermal constraints, efficiency cores are based on the Gracemont micro-architecture and can only run up to 3.40 GHz. When running integer benchmarks, the geometric mean performance of an E-core is $0.8x$ that of a P-core running at the same frequency (ROTEM et al., 2022). To understand when the memory latency advantage of the E-cores supersedes the performance advantage of P-cores, a benchmark was formulated, allowing us to explore the threshold where it is more beneficial to schedule threads on E-cores rather than P-cores, despite their performance differences.

```
void *consumer_thread(void *data) {
  consumerArguments *arguments = data;
  message *matrix = arguments->matrix;
  int *return_value = arguments->return_value;
  int acc = 0;
  int i = 0;
  while (i < MATRIX_SIZE - 1) {
    if (matrix[i].available) {
      accumulator += matrix[i].value;
      volatile int j = arguments->busy_wait;
      while(j--);
      i++;
    }
  }
  if (acc == matrix[MATRIX_SIZE - 1].value) {
    *return_value = true;
  } else {
    *return_value = false;
  }

  return NULL;
}
```

Listing 3.1 – Consumer thread source code

The benchmark is comprised of a producer and a consumer thread, where the producer posts random numbers to a queue in shared memory and tags them as available. The consumer takes those numbers when available and performs a busy wait incrementing a local counter, simulating a variable workload per memory operation. This allows us to control the ratio between memory and arithmetic operations and provides a run-time metric for performance evaluation.

```
void *producer_thread(void *data) {
  message *matrix = (message *)data;
  int acc = 0;
  srand(time(NULL));
  for (int i = 0; i < MATRIX_SIZE - 1; i++) {
    int temp = rand() % 10;
```

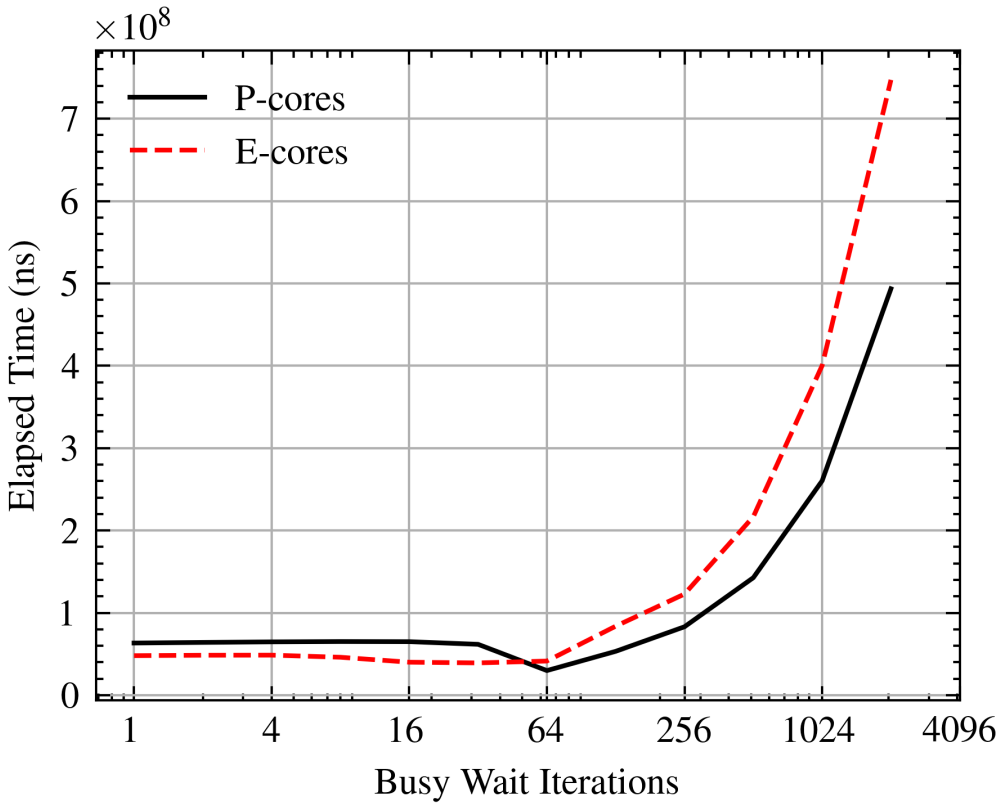
```
    matrix[i].value = temp;
    matrix[i].available = true;
    acc += temp;
  }
  matrix[MATRIX_SIZE - 1].value = acc;
  return NULL;
}
```

Listing 3.2 – Producer thread source code

In all cases, the *MATRIX_SIZE* was set to $1,000,000$, which provided an acceptable run-time in the order of hundreds of milliseconds suitable for profiling the application. The matrix was initialized to a random number to deliberately throw off the branch predictor. Each test case was run 100 times and averaged out.

We then follow by exploring the relationship between memory access and speedup. This is based on the hypothesis that the lower latency of the caches in the E-cores might benefit memory-bound workloads, as it is shared at the L2 level and has a smaller size.

Figure 3.2: Benchmark results (lower is better)



Source: The author

By varying the number of busy wait iterations, it was possible to generate a curve

transitioning the workload from being memory-bound (with fewer iterations) to CPU-bound (with more iterations). Using the Linux *perf* tool, the benchmark instruction distribution was measured using cores 0 and 2. For measuring the total number of retired micro-operations, the *inst_retired.any* event was used. To measure loads and stores, the *L1-dcache-loads* and *L1-dcache-stores* events were used respectively. In order to measure the execution time of the program, the *CLOCK_PROCESS_CPUTIME_ID* time source was used, which is tied to the *TSC* (Time Stamp Counter) high-resolution timer of the CPU. Only the execution of the matrix incrementing kernel was measured, removing any overhead of the support code.

Table 3.1: Benchmark instruction distribution

# Iterations	**Total Retired** (10^6)	**Load/stores** (10^6)	**%Mem**
1	17	8	48.5%
2	18	9	48.3%
4	19	9	47.8%
8	21	10	47.0%
16	24	11	46.1%
32	30	13	44.9%
64	44	19	43.3%
128	78	33	42.2%
256	140	58	41.0%
512	268	109	40.5%
1024	526	212	40.3%
2048	1,038	417	40.2%

Source: The author

As seen in Fig. 3.2, for a smaller relative number of CPU workload versus memory operations, the E-cores show a lower run time, despite being generally less performant than P-cores. However, as the benchmark becomes more CPU-bound, around the 32 iterations mark, the P-cores' raw speed advantage supersedes the higher memory latency and thus they show a decreased run-time relative to the E-cores.

This can be explained by the lower latency between the E-cores, as demonstrated in the characterization step. By taking advantage of the shared L2 cache, we avoid taking an expensive trip to the L3 cache, improving performance significantly when inter-process communication is the bottleneck. In this synthetic benchmark, we were able to obtain an average speedup of 43% when performing fewer than 32 busy wait operations.

To predict whether an application should be scheduled on performance or efficiency cores, we were able to find an average threshold of 44% memory micro-operations per total retired instructions, as per Table 3.1. This suggests that, when the thread retires fewer memory micro-operations, it is favorable to schedule it on performance cores.

Conversely, threads with higher loads and stores show increased shared memory communications, which suggests they should be scheduled on efficiency cores, despite lower general performance on integer workloads.

At the same time, the threshold for achieving the case of having better performance on E-cores is rather high. It requires an application with an almost purely memory-bound workload, with frequent inter-core communication that causes high traffic on the L2 cache. On most applications, this can be easily optimized by batching transactions between threads, causing this behavior to be, although verifiable in experiments, unsuitable as a heuristic for real-world task placement. Therefore, we will follow by collecting data from real-world workloads to better understand how tasks behave in each core micro-architecture.

3.3 Real-world workloads

As the benchmark proposed in the previous section is purposely synthetic, aiming to exercise a single characteristic of the system, we decided to also include some real-world workloads. To run this, a collection of workloads was selected from Phoronix Test Suite, an open-source software (LARABEL; TIPPETT, 2011). Several test profiles were selected and compiled into a test suite to simulate various usage scenarios. This was done aiming to construct an ample dataset for future developments. The complete set of benchmarks is available on Table 3.2.

To ensure the statistical significance of the measured data, Phoronix automatically repeats measurements until their standard deviation falls below a preset value. In this case, it was left at the default value of 3.50%, with a minimum of 3 runs per test. The final test result is comprised of the average of the runs. Processor execution was controlled by setting the CPU affinity to either only four P-cores (0, 2, 4, and 6, ignoring SMT siblings) or only four E-cores (8, 9, 10, 11, all in the same memory cluster). On benchmarks that supported doing so, we also passed down a flag reducing the thread count to match the available cores. All performance counter data was collected when running on a P-core, as they provided the most complete set of them.

During the benchmark runs, various performance counters were recorded using Linux's *perf* tool. The branch-related ones aimed to quantify how big the impact of the branch predictor is in a given workload. At the same time, we also measured the number of loads and misses at the first and level caches, in order to quantify how memory-bound

Table 3.2: Tests included in the test suite

Benchmark	Type	Description
system/gimp-1.1.3	Content Creator	Various image operations using the GIMP image editor
system/rawtherapee-1.0.1	Content Creator	Various RAW image operations using Rawtherapee
pts/blender-4.1.0	Content Creator	Rendering 3D scenes using Blender
pts/ffmpeg-7.0.1	Multimedia	Encoding video files using x264 and x265 codecs
pts/encode-mp3-1.7.4	Multimedia	Converting a WAV file to MP3
pts/vpxenc-3.2.0	Multimedia	Encoding a raw video file to VPX
pts/git-1.1.0	Developer	Completing various Git commands
pts/build-linux-kernel-1.16.0	Developer	Compiling the Linux kernel
pts/tensorflow-2.2.0	AI	Running inference on several small models
system/tesseract-ocr-1.0.1	AI	Converting images to text using Tesseract OCR
pts/rodinia-1.3.2	Benchmark	CPU benchmark using OpenMP
pts/dacapobench-1.1.0	Benchmark	Java CPU benchmark
pts/renaissance-1.3.0	Benchmark	Java JVM test suite
pts/himeno-1.3.0	Scientific	Linear solver of pressure Poisson
pts/stockfish-1.5.0	Gaming	Advanced open-source C++11 chess benchmark
pts/hackbench-1.0.0	Benchmark	Linux kernel stressor
pts/radiance-1.0.0	Scientific	NREL Radiance, a synthetic imaging system
pts/fftw-1.2.0	Scientific	Computes the discrete Fourier transform
system/octave-benchmark-1.0.1	Scientific	Completes several reference files via octave-benchmark
pts/mt-dgemm-1.2.0	Benchmark	Double General Matrix Multiply
pts/amg-1.1.0	Benchmark	Parallel algebraic multigrid solver for linear systems
pts/dolfyn-1.0.3	Scientific	Computational Fluid Dynamics
pts/cloverleaf-1.2.0	Benchmark	Lagrangian-Eulerian hydrodynamics benchmark
pts/minife-1.0.0	Scientific	Unstructured implicit finite element codes
pts/pennant-1.1.0	Scientific	Hydrodynamics on general unstructured meshes in 2D
pts/incompact3d-2.0.2	Scientific	Fortran-MPI based Navier-Stokes solver
pts/himeno-1.3.0	Scientific	Linear solver of pressure Poisson using a point-Jacobi method
pts/mrbayes-1.5.0	Scientific	Performs a bayesian analysis of a set of primate genome sequences
pts/mafft-1.6.2	Scientific	Performs an alignment of 100 pyruvate decarboxylase sequences

Source: The author (condensed from (Open Benchmarking., 2024))

an application is. The number of total cycles and instructions were measured to provide both CPI and a value against which other measurements could be normalized to avoid considering run time. Finally, a set of Top-Down Micro-architecture Analysis (TMA) events were selected. These were first proposed by (YASIN, 2014) and are now included in many recent Intel CPUs, aiming to account for common bottlenecks in super-scalar cores. A complete relation of the counters is available on Table 3.3. This set is similar to what was proposed by (SINGH; BHADAURIA; MCKEE, 2009), but focusing more heavily on the memory subsystem and branch predictor.

By collecting this data, a dataset can be built for a variety of workloads, allowing further analysis to take place aiming to correlate different performance counters to the performance gains achieved by placing a given task on a P-core instead of an E-core.

After running the Phoronix Test Suite with our custom test set, available in Table 3.2, we obtained the following results in Figure 3.3 for establishing the possible speed-up obtained on P-cores. The data is normalized against the E-cores, so bars to the right of the red line represent improvements while measurements to the left represent regressions. With this data, along with the performance counters, we are able to construct a speedup

Table 3.3: Recorded *perf* events/counters

Perf Counter	**Hardware Event**	**Description**
Branches	BR_INST_RETIRED.ALL_BRANCHES	Both taken and not taken branches
Branch Misses	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branches
L1d Loads	MEM_INST_RETIRED.ALL_LOADS	All memory loads
L1d Load Misses	L1D.REPLACEMENT	All memory loads not in L1 cache
L1i Load Misses	ICACHE_64B.IFTAG_MISS	Instruction loads not in L1 cache
Cache References	LONGEST_LAT_CACHE.REFERENCE	All Last Level Cache hits
Cache Misses	LONGEST_LAT_CACHE.MISS	All Last Level Cache misses
TMA Retiring	TOPDOWN_RETIRING.ALL	Slots used by issued µops that eventually get retired
TMA Mem Bound	TOPDOWN.MEMORY_BOUND_SLOTS	Execution stalls due to the memory subsystem
TMA Bad Spec	TOPDOWN.BAD_SPEC_SLOTS	Slots wasted due to incorrect speculations
TMA FE Bound	TOPDOWN_FE_BOUND.ALL	Slots when the frontend of the CPU undersupplies the backend
TMA BE Bound	TOPDOWN.BACKEND_BOUND_SLOTS	Slots when no µops are being delivered at the issue pipeline
Cycles	CPU_CLK_UNHALTED.THREAD	Number of cycles while the CPU was not halted
Instructions	INST_RETIRED.ANY	Number of retired instructions

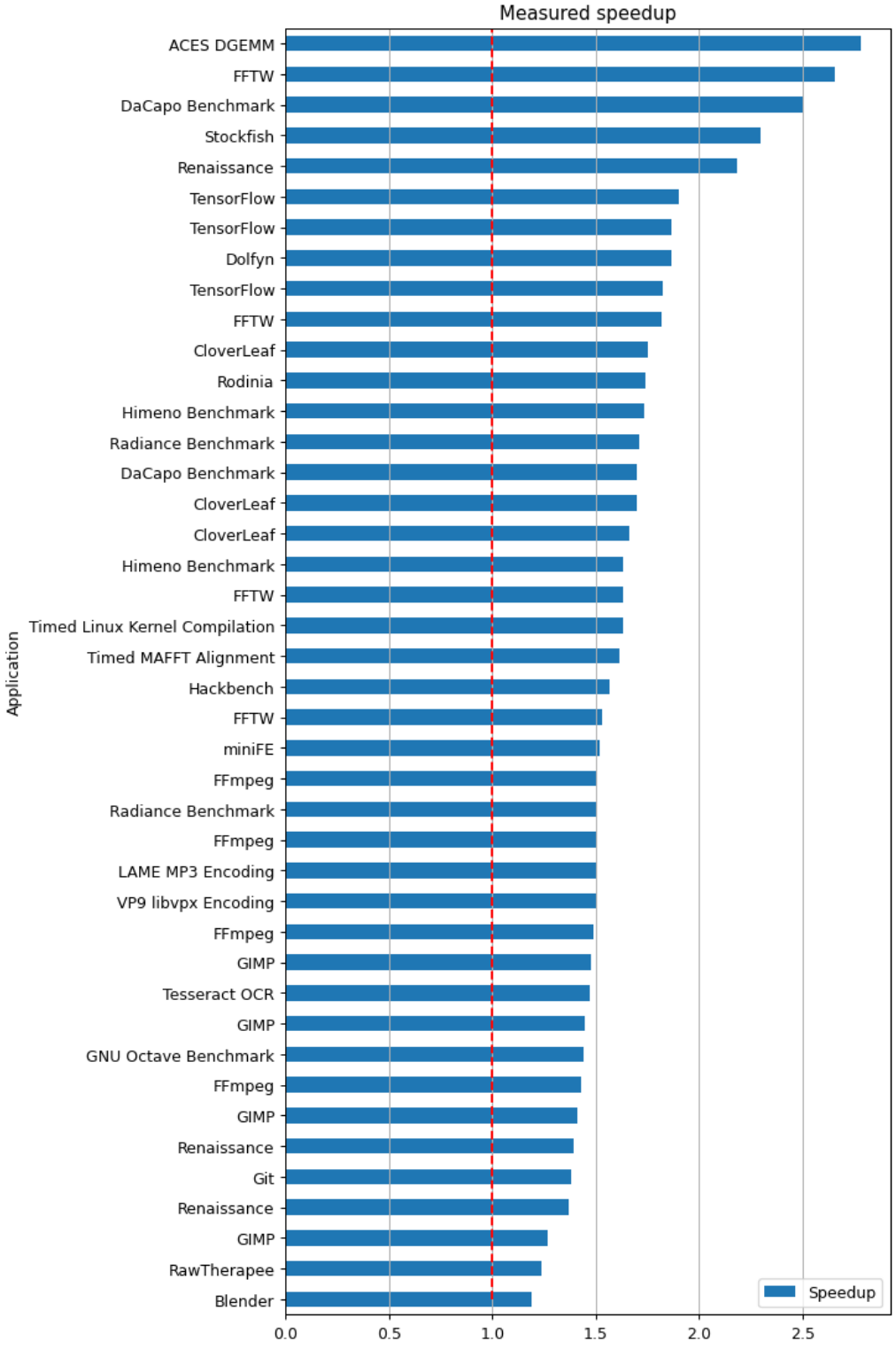
Source: The author (adapted from *perf* manual page)

factor (SF) figure for each workload, which is defined as $SF = \frac{Perf_P}{Perf_E}$, similar to what is done in (BILBAO; SAEZ; PRIETO-MATIAS, 2023). This SF will be used to determine which core is preferred to run each task, as it will maximize the potential throughput gain when multiple tasks are competing for the same CPU.

Overall, the geometric mean of the speed-up was that P-cores performed $1.64x$ better than E-cores, which is within the $1.76x$ at peak performance and $1.24x$ sustained figure read from the ACPI CPPC registers. As the benchmarking suite used is comprised of workloads of various durations, it is expected that we would fall within peak and sustained figures. As seen in Figure 3.4, most of the benchmarks fell at the expected range, with a few outliers reaching up to $2.78x$, as in the case of ACES DGEMM.

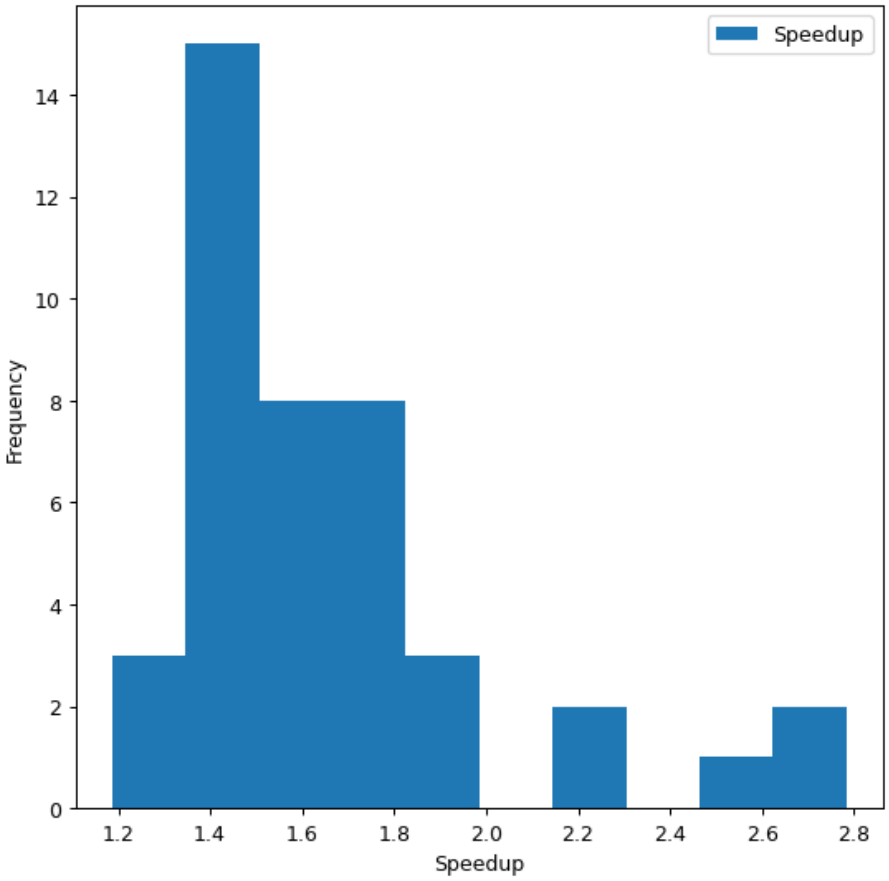
As seen in Figures 3.5 and 3.6, performance counter data by itself does not present any obvious structure to it. Due to this, a machine learning model was devised to make sense of the data collected.

Figure 3.3: Speedup per measured application



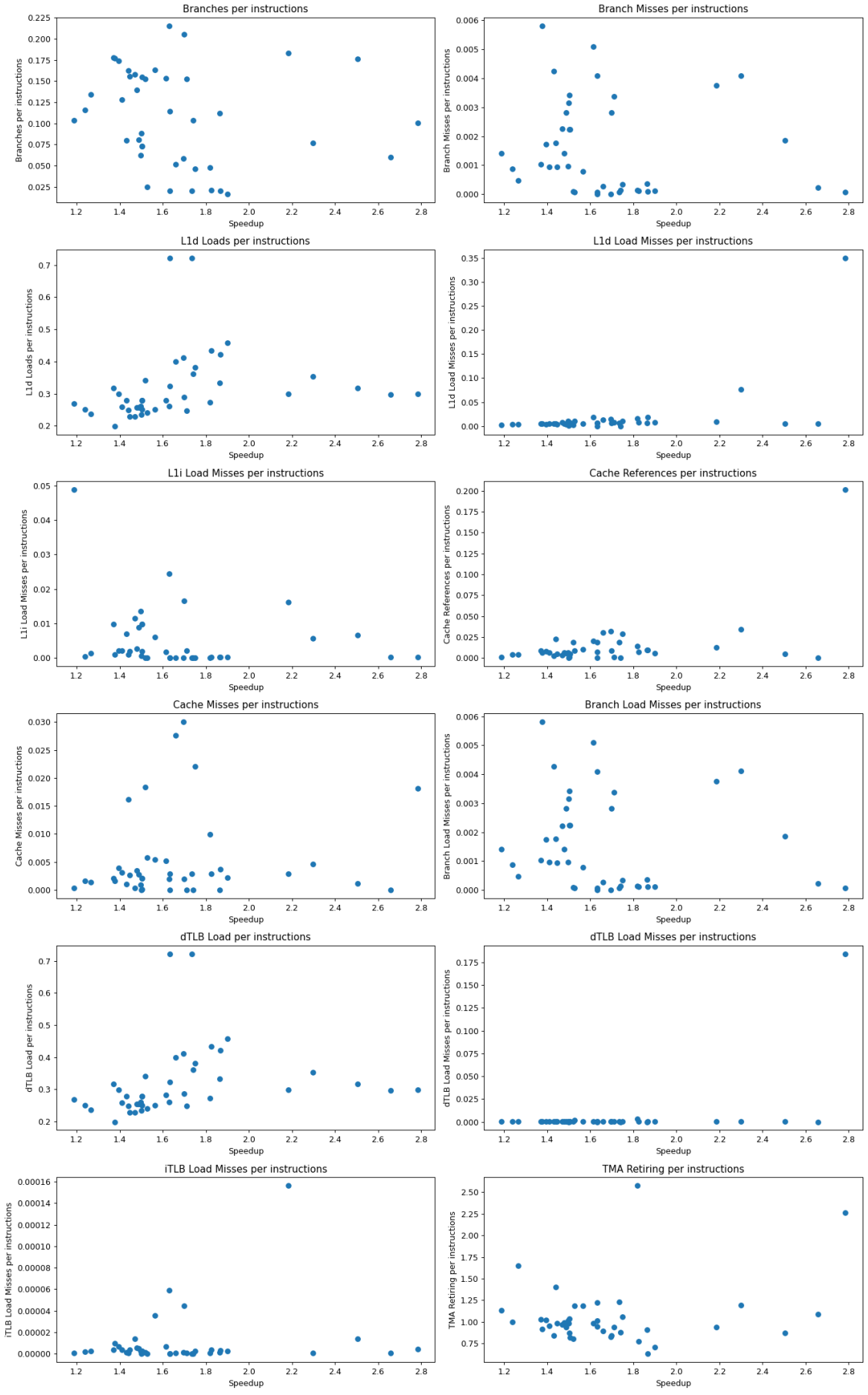
Source: The author

Figure 3.4: Distribution of the speedups



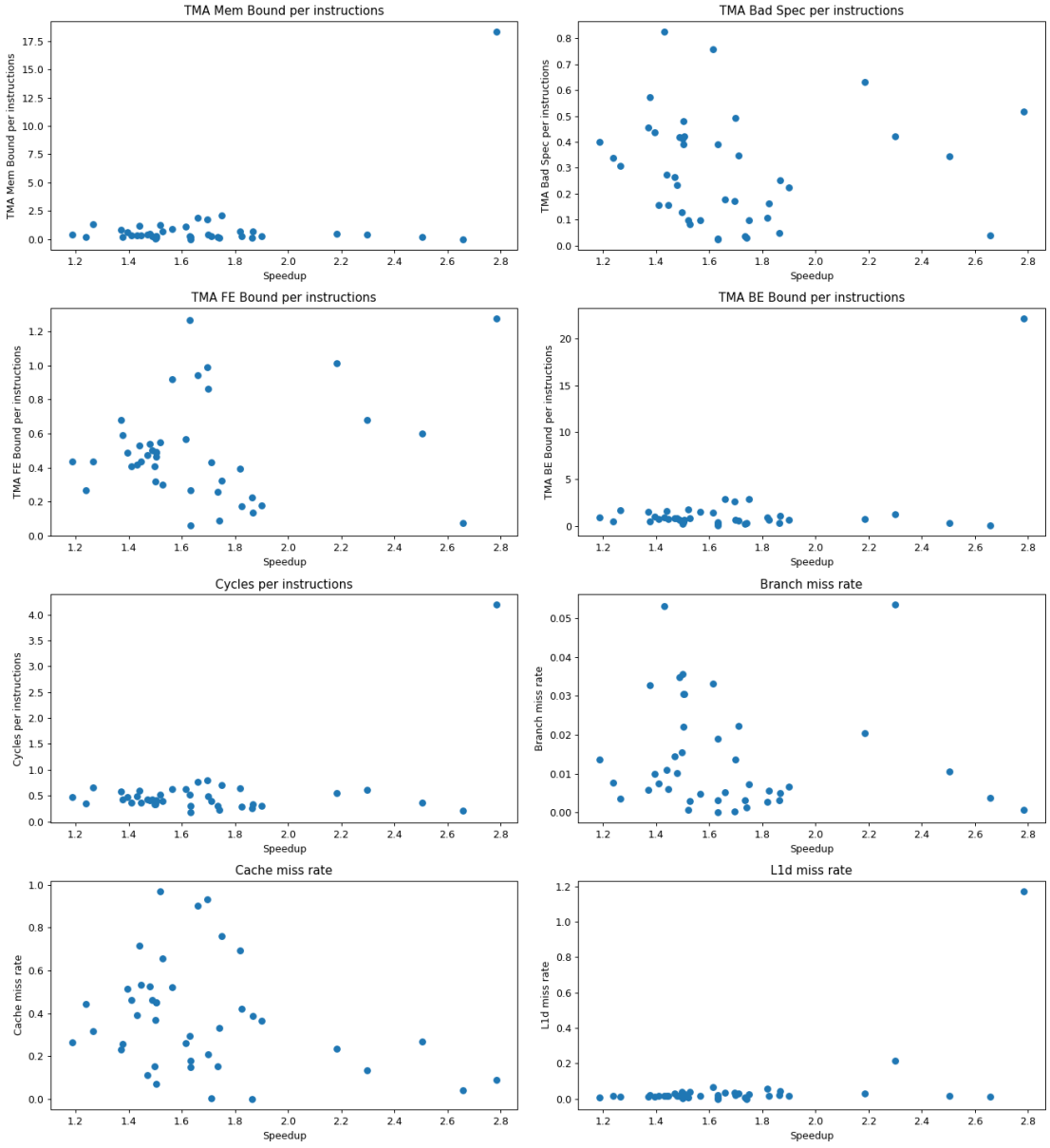
Source: The author

Figure 3.5: Performance counter data (part 1)



Source: The author

Figure 3.6: Performance counter data (part 2)

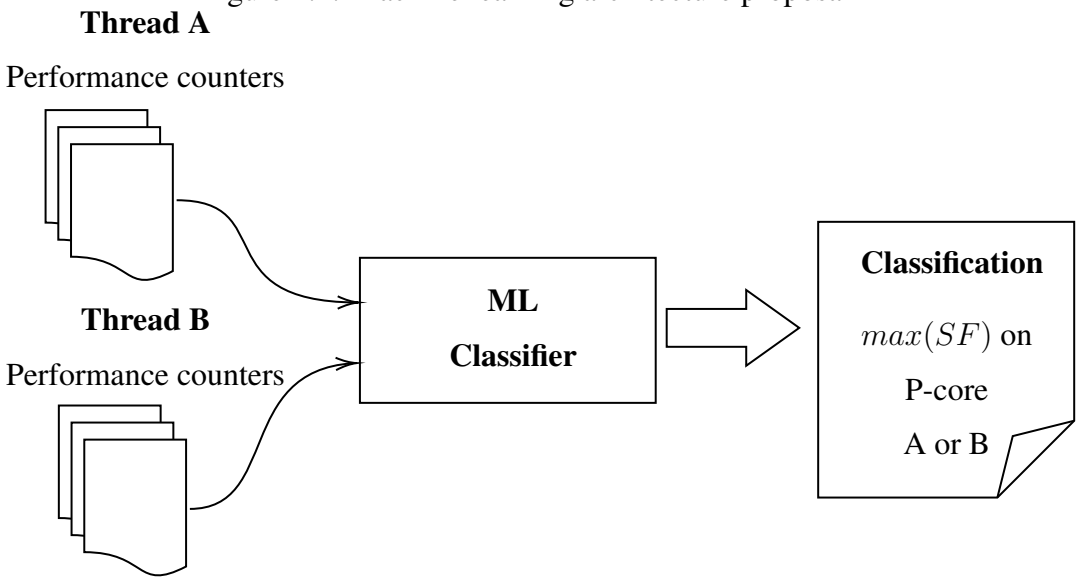


Source: The author

4 TASK-PLACEMENT PROPOSAL

To utilize the data collected in a way that could potentially be used to aid in scheduling decisions, a machine learning model was constructed. As the dataset of benchmarks is small, comprising only 42 runs, the model was structured as a classifier that takes as inputs the performance counters of two threads and outputs which of the two is most likely to benefit more from being allocated in a P-core. This allows for $42 \times 41 = 1722$ benchmark combinations to be generated, aiding in the learning of the model. The proposed model can be seen in Figure 4.1.

Figure 4.1: Machine learning architecture proposal



Source: The author

The model is then implemented using a Gradient-Boosting algorithm in the *scikit-learn* Python library (PEDREGOSA et al., 2011), using the *GradientBoostingClassifier* class. This algorithm was selected due to its performance on sparse tabular data and good overfitting rejection (NATEKIN; KNOLL, 2013). All hyper-parameters were left as the library default. The dataset was first normalized using the *MinMaxScaler*, which translated every feature to be within the $[0, 1]$ range, balancing the feature set. Then it was split, with 80% being used for training and 20% for testing, with the results seen in Table 4.1. Finally, the model was tested again using K-fold cross-validation with $K = 5$, achieving 91.1% accuracy with a standard deviation of 4.7%. Overall, the mean squared error on the test set was 0.0722. This is significantly above the 50% accuracy of a random binary classifier.

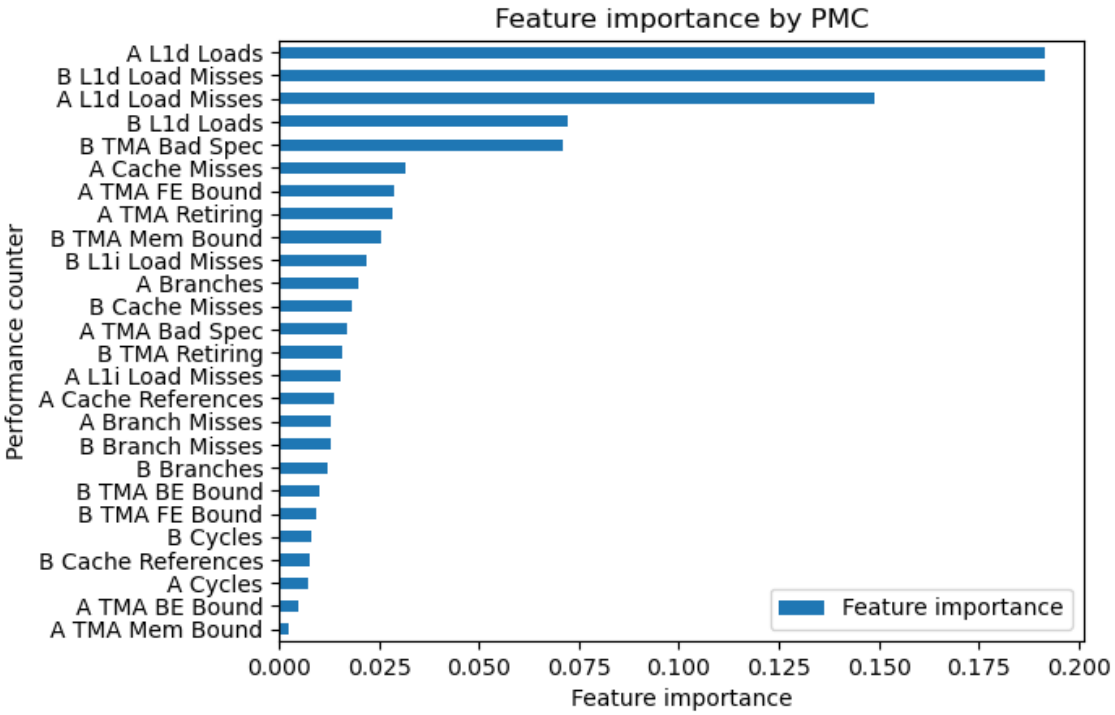
Table 4.1: Classification report on the test set

	precision	**recall**	**f1-score**	**support**
0.0	0.93	0.94	0.94	210
1.0	0.92	0.91	0.91	150
accuracy	0.93	360		
macro avg	0.93	0.92	0.93	360
weighted avg	0.93	0.93	0.93	360

Source: The author

As the Gradient Boosting algorithm used is comprised of multiple smaller decision trees, it is possible to examine the contribution of each one. By backtracking on the weights, it is possible to determine how heavily each feature influences the final decision. This is done using the *feature_importances_* attribute of the classifier exposed in *scikit-learn*. This is based on the impurity of each feature and is calculated as the normalized total reduction of the criterion brought by that feature (PEDREGOSA et al., 2011). The higher the value, the higher the feature importance.

Figure 4.2: Feature importance of the Gradient Boosting Decision Tree



Source: The author

As seen in Figure 4.2, the classifier is dominated by performance counters influenced by the memory subsystem. Both threads' memory activity is used to evaluate

the classification. This is in line with the measurements provided in the earlier section, where we analyzed the core-to-core latency. The model values highly not only the memory activity in general, but the load misses from the L1 cache as well. This is interesting as the E-cores have $2/3$ of the data cache of the P-cores ($32kB$ vs $48kB$), but a much larger, though shared, L2 ($2048kB$ shared between E-cores vs. $1280kB$ per P-core). At the same time, despite the E-cores having an L1 instruction cache twice as large as the P-cores ($64kB$ vs $32kB$), the difference has much less impact on the performance as per the model.

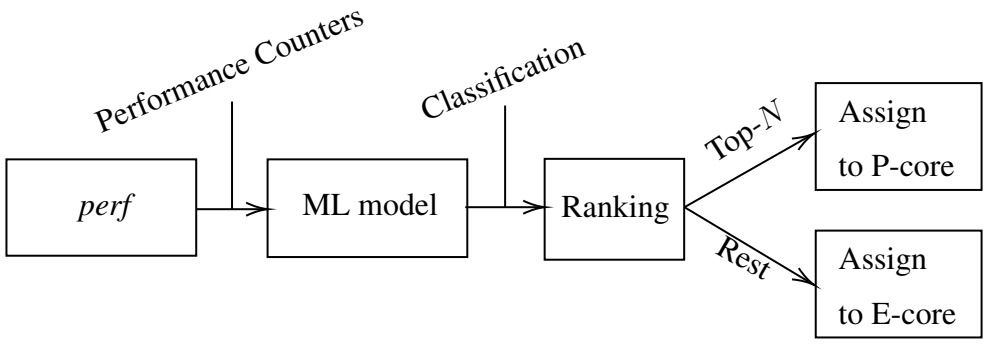
Below the memory subsystem, we also see a hint of the branch predictor metrics being used in the form of the TMA counters. For instance, *B TMA Bad Spec* ranks just below memory data loads/misses and is indicative of the simplistic nature of the E-core's branch predictor causing performance slots due to stalls caused by bad speculation events.

Finally, in the future the feature importance matrix could also be used to further optimize the classifier to reduce the number of required inputs. This would reduce the online profiling overhead and model complexity, leading to a more streamlined classifier. At the present, the complete model was exported in *pickle* format to be used in the next section.

4.1 Tournament-style scheduler overlay

At last, we can build the scheduler using the knowledge and models obtained from previous sections. As the model was built to discriminate between two threads, the implementation of the core pinning algorithm was done in a tournament-style classifier. At its core, it maintains a list of top N preferred threads to occupy P-cores, with N being defined further down. That is, we construct a ranking based on the relative ordering provided by the machine learning model. Based on that ranking, it then pins threads to the appropriate cores, as seen in Figure 4.3. This way, when multiple threads are competing for CPU, we will delegate the fastest cores to the threads which will benefit the most, while demoting other threads to efficiency cores as to free up the performance ones. This way we can maximize the utility of the performance cores while minimizing degradation caused by the efficient ones.

Figure 4.3: Topology-aware scheduler model



Source: The author

This is implemented using a *Python* script that periodically reads data fed from Linux's *perf* tool. By invoking `perf stat -e <counters> --per-thread -x, -I<interval> --interval-count 1`, we are able to easily monitor performance counters across the whole system. The `-e` flag allows us to specify which events to read and the `--per-thread` flag indicates that we want this data to be discriminated, instead of aggregate. This data is provided in tab-separated value (TSV) format as indicated by the `-x,` flag and is parsed in real-time, feeding the machine learning model. Every invocation provides a single snapshot of data, as indicated by the `-I<interval> --interval-count 1` flags. This is done as the `--per-thread` flag causes *perf* to only monitor threads already existing during the command's invocation. Finally, the script then uses the *cpuset* API to place threads in the correct cores by setting the appropriate CPU affinity of each. The complete script is available in the Appendix.

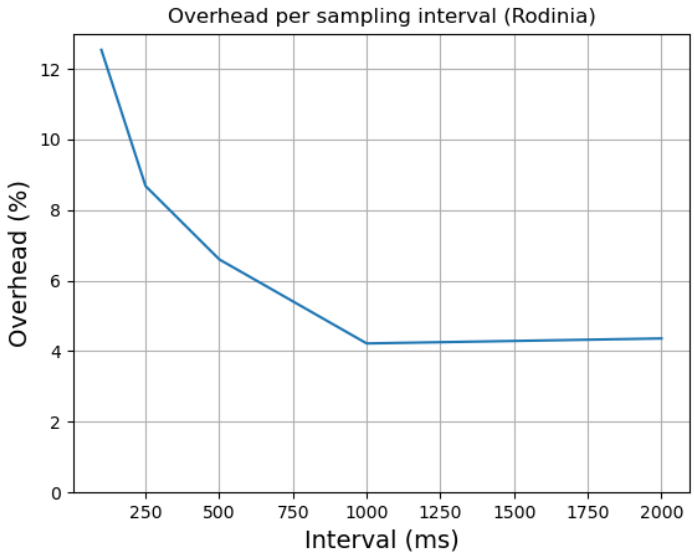
```
    def __gt__(self, other):
        my_data = [self.counters.get(f)/self.counters.get("cpu_core/instructions/") for f in FEATURES]
        other_data = [other.counters.get(f)/other.counters.get("cpu_core/instructions/") for f in FEATURES]
        scaled = scaler.transform([my_data + other_data])
        prediction = clf.predict(scaled)
        return prediction[0] == 1.0
```

Listing 4.1 – Process comparison operator

As we have built a classifier, we must find a way to build the ranking by using the model to perform a comparison operation. We do this by building a `Process` class and overriding Python's `__gt__` method. This defines a relative ordering, which is then used by the `functools.total_ordering` decorator to provide a total ordering. This enables the data to be sorted using Python's `sorted` function, which utilizes *Timsort* under the hood (DALKE; HETTINGER, 2024). This sorting algorithm has a complexity between

$O(n \log n)$ and $O(n)$, so the model has to be executed very closely to the least possible. As in the data collection stage, we normalize every metric by the retired instruction count, making every feature invariant of the sampling rate.

Figure 4.4: perf overhead per sampling interval



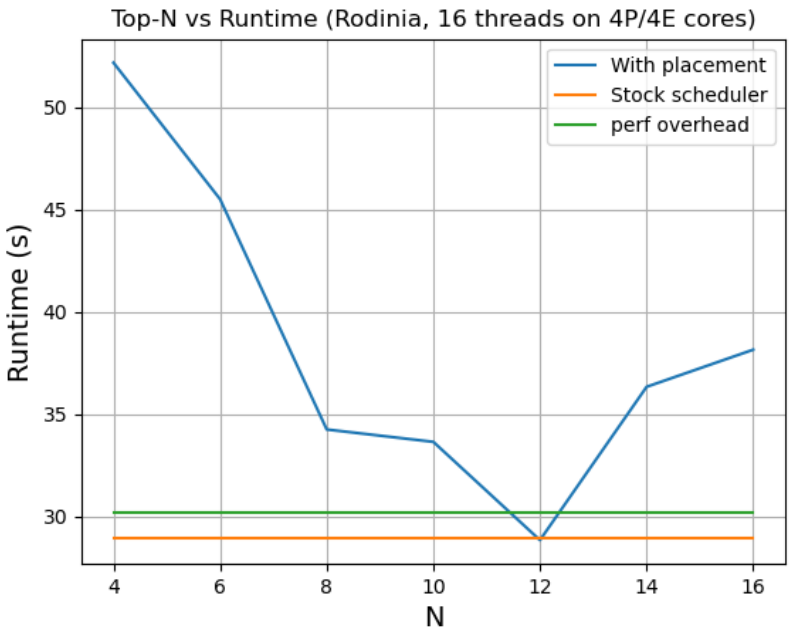
Source: The author

A CPU utilization cutoff was defined to avoid applying the model on low-impact threads and to reduce noise. At the current state, threads with less than 5% total CPU utilization are ignored by the task placer. Using the gradient descent model for inference using *sklearn*, the script takes approximately $10.59ms$ to sort 16 threads, averaging at $662\mu s$ per thread, accounting for approximately 1.6% CPU usage. At the same time, the *perf* command is spawned every second, taking up to an average of 10% of a single CPU core when monitoring 16 threads, causing the overhead shown in Figure 4.4. As the time scale of the applications being run was multiple minutes, the refresh rate of the algorithm was set at $1000ms$, as this provided a reasonable balance between overhead and fast settling. However, it could potentially run much faster to accommodate shorter-lived threads and take advantage of shorter run phases at a higher cost. It is interesting to note that the largest overhead does not come from the model itself, but from the *perf* utility. That strongly suggests that the overhead could be reduced significantly if performance counters were read directly from the kernel, without an intermediate tool.

4.2 Performance evaluation

As one of the core parameters of our model is a ranking, we started by evaluating for which N the Top-N allocator performed the best. At first, it was estimated that the N for the Top-N ranking should be equal to the number of P-cores available in the system. That is, every P-core should only handle a single thread. This was expected to be reasonable as the benchmarks in the study utilized very closely to 100% CPU. However, by looking at how Linux scheduled tasks, it became quickly apparent that delegating too little threads to the P-cores would lead to E-cores being overworked. Therefore, we experimented with oversubscribing P-cores, as an attempt to lower the burden on the lower-performing efficiency ones, achieving the results available in Figure 4.5. For the evaluation, we utilized *Rodinia* as it is a well-behaved OpenMP benchmark that can run with a configurable amount of threads. The number of threads was set to 16, deliberately oversubscribing the processor, which was configured to disable simultaneous multithreading (Intel Hyper-Threading) and had one of the E-core clusters disabled as well, leaving 4 true P-cores and 4 E-cores online for a total amount of 8 threads. Therefore, our CPU was oversubscribed with a factor of $2\times$ threads in all of the following benchmarks.

Figure 4.5: Top-N vs Performance Evaluation (Rodinia CFD)

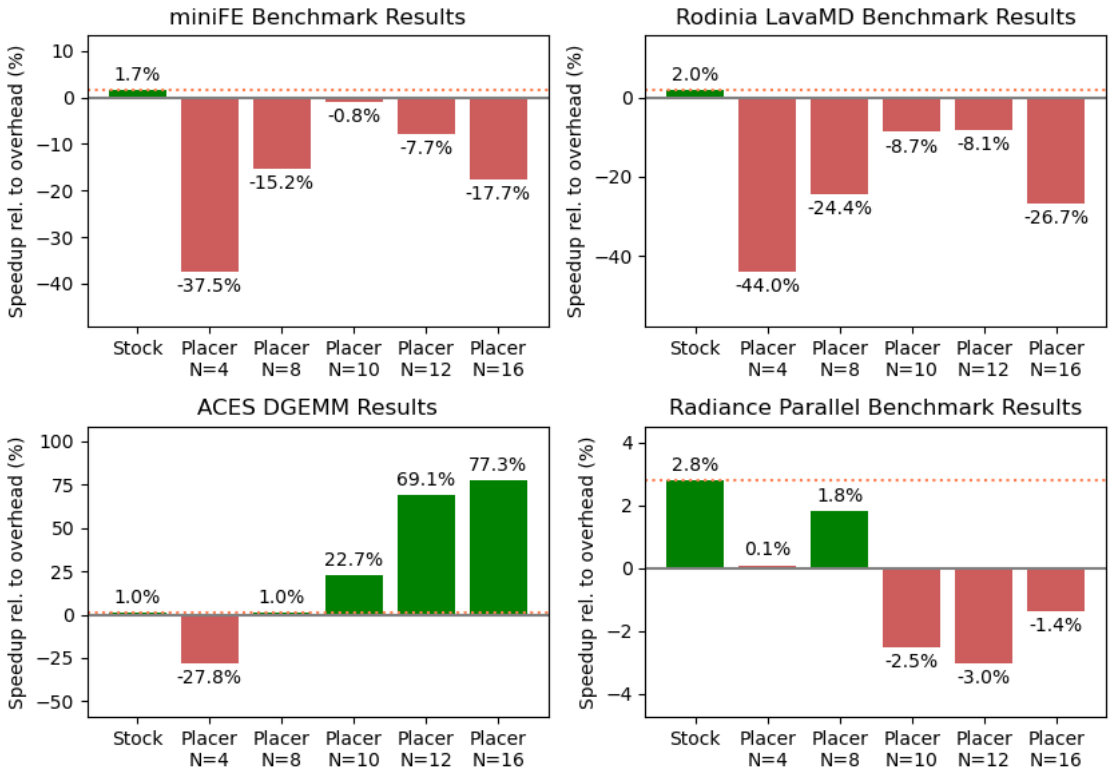


Source: The author

As seen in Figure 4.5, the best performance happened when we oversubscribed

the P-cores by a factor of $2.5\times$, allocating 12 threads to the 4 physical cores, as seen in the blue line. In the orange line, we can see the runtime of the benchmark using the stock scheduler, while in green we can see the baseline performance of our scheduler. That is, the performance with *perf* collecting data and the model running, but without performing any placement. As seen in the case $N = 12$, we are able to achieve a speed-up of 4.44% compared to the overhead baseline. Oversubscribing the performance cores further degrades performance.

Figure 4.6: Multi-benchmark performance evaluation



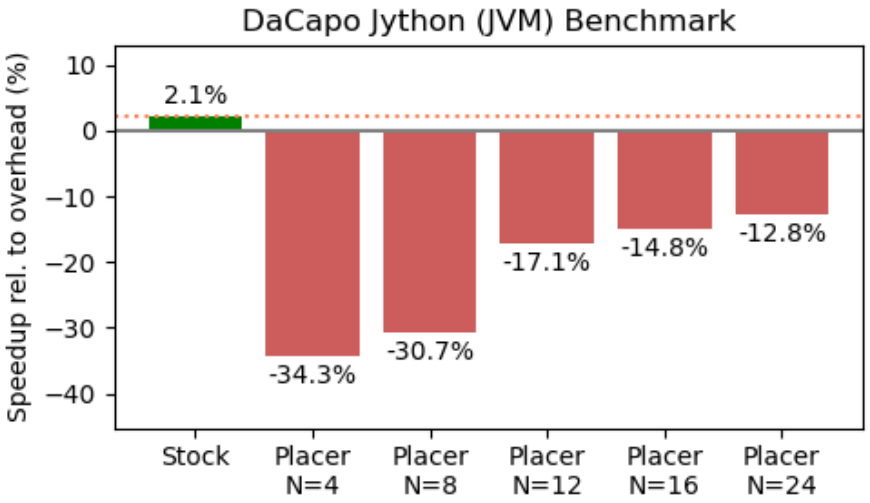
Source: The author

For other benchmarks, the result was not as clear, as seen in Figure 4.6. *miniFE* and *Rodinia LavaMD* presented a degraded performance in all test cases, with the least amount of degradation happening around $N = 10$ and $N = 12$, similar to what was evaluated before. At the same time, we won in performance in two benchmarks. In *ACES DGEMM*, we were able to improve performance by up to 77.3% in the case of $N = 16$, which means all threads were allocated to P-cores. Unfortunately, this also means that the model was not used in that case. However, when lower N numbers were used a speedup was still noted, reaching up to 69.1% in the $N = 12$ case, which still left 4 threads to the 4

remaining E-cores. Finally, with *Radiance* we saw an increase in performance compared to the overhead line, in the $N = 8$ case, which is significantly lower than the other cases. These four benchmarks combined with the one presented before result in a geometric mean of 7.76% speedup.

Finally, the performance of Java applications was also analyzed. This is important as we could not control the number of threads of these benchmarks, only the number of workers in the fork-join pool. That means the placer got stressed with > 35 threads in benchmarks such as *DaCapo Benchmark*. These threads range from proper workers in the fork-join pool to garbage collecting and other management operations.

Figure 4.7: Java Virtual Machine performance evaluation



Source: The author

As seen in Figure 4.7, the proposed model did not cope well with the fragmented mode of execution of Java code, causing substantial slowdowns regardless of the thread allocation distribution. This is hypothesized to be caused by the unpredictable nature of Java code. For instance, the thread placer is running every $1000ms$, however, the garbage collector runs frequently for much smaller time frames. Unfortunately, this causes the model to fail as it does not react quickly enough to manage this kind of bursty workload.

5 CONCLUSION AND FUTURE WORK

The experimental investigation conducted throughout this thesis has examined the scheduling strategies and optimizations for heterogeneous architectures, specifically focusing on the Intel Core i7-1260P processor and the Alder Lake generation. We were able to verify experimentally that some synthetic workloads can benefit from being scheduled in E-cores, despite their lower performance, due to the lower-latency memory architecture. When there is intense inter-core communication, up to 43% speedup can be realized that way.

Furthermore, a dataset was built using 42 benchmarks and collecting several performance counters. They were then used to train a machine learning model based on gradient boost, which classifies thread placement for pairs of competing threads. Using K-fold cross-validation, we were able to achieve 91.1% accuracy on our dataset. This dataset was then used in a script to modify the thread placement in a running system. On select benchmarks, we were able to achieve up to 69.9% speedup, with a geometric average of 7.76%. Finally, we analyzed outliers that regressed using our approach, identifying the high number of bursty threads as the issue behind our model's poor performance.

In the future, a possible way to reduce experiment overhead would be ditching the *Python* and *perf* solution and start collecting data directly inside the kernel. To do so, a possible route would be implementing a loadable online profiling module using eBPF and overriding part of the scheduler using the *sched_ext* framework, which should land in the kernel version 6.11. As for the machine learning model, it could be reorganized to collect different phases of the applications, as some benchmarks have significant phase variability. This would also lead to more data points being collected, enhancing the usefulness of the machine learning techniques applied. Finally, other performance counters could be experimented on, as we only analyzed a small amount of them in this work.

REFERENCES

ALPAYDIN, E. **Introduction to Machine Learning**. 3. ed. Cambridge, MA: MIT Press, 2014. (Adaptive Computation and Machine Learning). ISBN 978-0-262-02818-9.

Apple Inc. **Apple unleashes M1**. 2020. Https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

ARM. **Arm Big.LITTLE**. 2011. Https://www.arm.com/technologies/big-little.

ARM. **Arm DynamIQ: Technology for the next era of compute**. 2017. Https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-dynamiq-technology-for-the-next-era-of-compute.

BILBAO, C.; SAEZ, J. C.; PRIETO-MATIAS, M. Flexible system software scheduling for asymmetric multicore systems with pmcsched: A case for intel alder lake. **Concurrency and Computation: Practice and Experience**, v. 35, n. 25, p. e7814, 2023. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.7814>.

BORAN, N. K.; YADAV, D. K.; IYER, R. Classification based scheduling in heterogeneous isa architectures. In: **2020 24th International Symposium on VLSI Design and Test (VDAT)**. [S.l.: s.n.], 2020. p. 1–6.

BORKAR, S. Thousand core chips: a technology perspective. In: **Proceedings of the 44th Annual Design Automation Conference**. New York, NY, USA: Association for Computing Machinery, 2007. (DAC '07), p. 746–749. ISBN 9781595936271. Available from Internet: <https://doi.org/10.1145/1278480.1278667>.

CRAEYNEST, K. V. et al. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In: **2012 39th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2012. p. 213–224.

DALKE, A.; HETTINGER, R. **Sorting Techniques**. 2024. Available from Internet: <https://docs.python.org/3/howto/sorting.html>.

DAS, R. et al. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In: **2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.: s.n.], 2013. p. 107–118.

FRIEDMAN, J. H. Greedy function approximation: A gradient boosting machine. **The Annals of Statistics**, Institute of Mathematical Statistics, v. 29, n. 5, p. 1189 – 1232, 2001. Available from Internet: <https://doi.org/10.1214/aos/1013203451>.

Intel Corp. **Media Presentation: Intel Core Ultra Processors**. Https://download.intel.com/newsroom/2023/ai/ai-everywhere-2023/Intel-Core-Ultra-Processors-Media-Presentation.pdf.

Intel Corp. **12th Gen Intel® Core™ Mobile Processor Product Brief**. 2021. Https://download.intel.com/newsroom/2022/ces2022/12th-gen-intel-mobile-product-brief.pdf.

Intel Corp. **Intel® Core™ i7-1260P Processor**. [S.l.], 2022. Https://www.intel.com/content/www/us/en/products/sku/226254/intel-core-i71260p-processor-18m-cache-up-to-4-70-ghz/specifications.html.

Intel Corp. **12th Generation Intel® Core™ Processors, Volume 1 of 2**. [S.l.], 2023. Https://www.intel.com/content/www/us/en/content-details/655258/12th-generation-intel-core-processors-datasheet-volume-1-of-2.html.

Intel Corp. **Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4**. [S.l.], 2023. Https://cdrdv2.intel.com/v1/dl/getContent/671200.

JIANG, Y. et al. Array regrouping on cmp with non-uniform cache sharing. In: COOPER, K.; MELLOR-CRUMMEY, J.; SARKAR, V. (Ed.). **Languages and Compilers for Parallel Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. p. 92–105. ISBN 978-3-642-19595-2.

KANDEMIR, M. et al. Cache topology aware computation mapping for multicores. In: **Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery, 2010. (PLDI '10), p. 74–85. ISBN 9781450300193. Available from Internet: <https://doi.org/10.1145/1806596.1806605>.

KUMAR, R. et al. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In: **Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.** [S.l.: s.n.], 2003. p. 81–92.

LARABEL, M.; TIPPETT, M. Phoronix test suite. **Phoronix Media,[Online]. Available: http://www. phoronix-test-suite. com/.[Accessed May 2024]**, 2011.

LOCUZA. **Die walkthrough: Alder Lake-S/P and a touch of Zen 3**. 2022. Available from Internet: <https://locuza.substack.com/p/die-walkthrough-alder-lake-sp-and>.

MOHAMED, A. M.; MUBARK, N.; ZAGLOUL, S. Performance aware shared memory hierarchy model for multicore processors. **Scientific Reports**, v. 13, n. 1, p. 7313, May 2023. ISSN 2045-2322. Available from Internet: <https://doi.org/10.1038/s41598-023-34297-3>.

MUTLU, O.; MEZA, J.; SUBRAMANIAN, L. The main memory system: Challenges and opportunities. **Communications of the Korean Institute of Information Scientists and Engineers**, Korean Institute of Information Scientists and Engineers, v. 33, n. 2, p. 16–41, 2015.

NATEKIN, A.; KNOLL, A. Gradient boosting machines, a tutorial. **Frontiers in Neurorobotics**, v. 7, 2013. ISSN 1662-5218. Available from Internet: <https://www.frontiersin.org/journals/neurorobotics/articles/10.3389/fnbot.2013.00021>.

Open Benchmarking. **Open Benchmarking**. 2024. Https://openbenchmarking.org/.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011.

RIGTORP, E. **C2Clat**. [S.l.], 2020. Https://github.com/rigtorp/c2clat.

ROTEM, E. et al. Intel alder lake cpu architectures. **IEEE Micro**, v. 42, n. 3, p. 13–19, 2022.

SAEZ, J. C. et al. PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler. **The Computer Journal**, v. 60, n. 1, p. 60–85, 01 2017. ISSN 0010-4620. Available from Internet: <https://doi.org/10.1093/comjnl/bxw065>.

SAEZ, J. C.; PRIETO-MATIAS, M. Evaluation of the intel thread director technology on an alder lake processor. In: **Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems**. New York, NY, USA: Association for Computing Machinery, 2022. (APSys '22), p. 61–67. ISBN 9781450394413. Available from Internet: <https://doi.org/10.1145/3546591.3547532>.

SHEIKH, S. Z.; PASHA, M. A. Energy-efficient cache-aware scheduling on heterogeneous multicore systems. **IEEE Transactions on Parallel and Distributed Systems**, v. 33, n. 1, p. 206–217, 2022.

SINGH, K.; BHADAURIA, M.; MCKEE, S. A. Real time power estimation and thread scheduling via performance counters. **SIGARCH Comput. Archit. News**, Association for Computing Machinery, New York, NY, USA, v. 37, n. 2, p. 46–55, jul 2009. ISSN 0163-5964. Available from Internet: <https://doi.org/10.1145/1577129.1577137>.

STOICA, I.; ABDEL-WAHAB, H. **Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation**. USA, 1995.

THARWAT, A. Classification assessment methods. **Applied Computing and Informatics**, Emerald, v. 17, n. 1, p. 168–192, jul. 2020. ISSN 2210-8327. Available from Internet: <http://dx.doi.org/10.1016/j.aci.2018.08.003>.

TORVALDS, L. **The Linux Kernel**. [S.l.], 2024. 6.8.0.

YASIN, A. A top-down method for performance analysis and counters architecture. In: **2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.: s.n.], 2014. p. 35–44.

ZHANG, E. Z.; JIANG, Y.; SHEN, X. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In: **Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: Association for Computing Machinery, 2010. (PPoPP '10), p. 203–212. ISBN 9781605588773. Available from Internet: <https://doi.org/10.1145/1693453.1693482>.

ZHANG, Y.; KANDEMIR, M.; YEMLIHA, T. Studying inter-core data reuse in multicores. In: **Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems**. New York, NY, USA: Association for Computing Machinery, 2011. (SIGMETRICS '11), p. 25–36. ISBN 9781450308144. Available from Internet: <https://doi.org/10.1145/1993744.1993748>.

APPENDIX — MODIFICATIONS TO THE PHORONIX TEST SUITE

```
--- a/pts-core/modules/linux_perf.php
+++ b/pts-core/modules/linux_perf.php
@@ -71,7 +71,7 @@ class linux_perf extends pts_module_interface
                // Set the perf command to pass in front of all tests to run
                self::$tmp_file = tempnam(sys_get_temp_dir(), 'perf');
                // -d or below is more exhaustive list
-               $test_run_request->exec_binary_prepend = 'perf stat -e branches,branch-misses,cache-misses,cache-references,cycles,instructions,cs,cpu-clock,page-faults,duration_time,task-clock,L1-dcache-load-misses,L1-dcache-loads,L1-dcache-prefetches,L1-icache-load-misses,context-switches,cpu-migrations,branch-loads,branch-load-misses,dTLB-loads,dTLB-load-misses,iTLB-load-misses,iTLB-loads -o ' . self::$tmp_file . ' ';
+               $test_run_request->exec_binary_prepend = 'perf stat -e branches,branch-misses,cache-misses,cache-references,cycles,instructions,cs,cpu-clock,page-faults,duration_time,task-clock,L1-dcache-load-misses,L1-dcache-loads,L1-icache-load-misses,context-switches,cpu-migrations,branch-loads,branch-load-misses,dTLB-loads,dTLB-load-misses,iTLB-load-misses -M tma_core_bound -o ' . self::$tmp_file . ' ';
        }
        public static function __post_test_run_success($test_run_request)
        {
@@ -98,22 +98,28 @@ class linux_perf extends pts_module_interface
                                'page-faults' => array('Page Faults', 'Faults', 'LIB'),
                                'context-switches' => array('Context Switches', 'Context Switches', 'LIB'),
                                'cpu-migrations' => array('CPU Migrations', 'CPU Migrations', 'LIB'),
-                               'branches' => array('Branches', 'Branches', ''),
-                               'branch-misses' => array('Branch Misses', 'Branch Misses', 'LIB'),
+                               'cpu_core/branches' => array('Branches', 'Branches', ''),
+                               'cpu_core/branch-misses' => array('Branch Misses', 'Branch Misses', 'LIB'),
                                'seconds user' => array('User Time', 'Seconds', 'LIB'),
                                'seconds sys' => array('Kernel/System Time', 'Seconds', 'LIB'),
-                               'stalled-cycles-frontend' => array('Stalled Cycles Front-End', 'Cycles Idle', 'LIB'),
-                               'stalled-cycles-backend' => array('Stalled Cycles Back-End', 'Cycles Idle', 'LIB'),
-                               'L1-dcache-loads' => array('L1d Loads', 'L1d Cache Loads', ''),
-                               'L1-icache-loads' => array('L1i Loads', 'L1i Cache Loads', ''),
-                               'L1-dcache-load-misses' => array('L1d Load Misses', 'L1 Data Cache Load Misses', 'LIB'),
-                               'L1-icache-load-misses' => array('L1i Load Misses', 'L1 Instruction Cache Load Misses', 'LIB'),
-                               'cache-misses' => array('Cache Misses', 'Cache Misses',
```

```
    'LIB'),
-                               'branch-load-misses' => array('Branch Load Misses', '
    Branch Load Misses', 'LIB'),
-                               'dTLB-load-misses' => array('dTLB Load Misses', 'dTLB
    Load Misses', 'LIB'),
-                               'ex_ret_mmx_fp_instr.sse_instr' => array('SSE
    Instructions', 'SSE Instructions', ''),
-                               'fp_ret_sse_avx_ops.all' => array('SSE+AVX Instructions'
    , 'AVX Instructions', ''),
-                               'instructions' => array('Instructions', 'Instructions',
    'LIB'),
+                               'cpu_core/L1-dcache-loads' => array('L1d Loads', 'L1d
    Cache Loads', ''),
+                               'cpu_core/L1-icache-loads' => array('L1i Loads', 'L1i
    Cache Loads', ''),
+                               'cpu_core/L1-dcache-load-misses' => array('L1d Load
    Misses', 'L1 Data Cache Load Misses', 'LIB'),
+                               'cpu_core/L1-icache-load-misses' => array('L1i Load
    Misses', 'L1 Instruction Cache Load Misses', 'LIB'),
+                               'cpu_core/cache-references' => array('Cache References',
     'Cache References', 'LIB'),
+                               'cpu_core/cache-misses' => array('Cache Misses', 'Cache
    Misses', 'LIB'),
+                               'cpu_core/branch-load-misses' => array('Branch Load
    Misses', 'Branch Load Misses', 'LIB'),
+                               'cpu_core/dTLB-load' => array('dTLB Load', 'dTLB Load',
    'LIB'),
+                               'cpu_core/dTLB-load-misses' => array('dTLB Load Misses',
     'dTLB Load Misses', 'LIB'),
+                               'cpu_core/iTLB-load-misses' => array('iTLB Load Misses',
     'iTLB Load Misses', 'LIB'),
+                               'cpu_core/TOPDOWN.slots' => array('Core Bound Slots', '
    Core Bound Slots', ''),
+                               'cpu_core/topdown-retiring' => array('TMA Retiring', '
    TMA Retiring', ''),
+                               'cpu_core/topdown-mem-bound' => array('TMA Mem Bound', '
    TMA Mem Bound', ''),
+                               'cpu_core/topdown-bad-spec' => array('TMA Bad Spec', '
    TMA Bad Spec', ''),
+                               'cpu_core/topdown-fe-bound' => array('TMA FE Bound', '
    TMA FE Bound', ''),
+                               'cpu_core/topdown-be-bound' => array('TMA BE Bound', '
    TMA BE Bound', ''),
+                               'cpu_core/instructions' => array('Instructions', '
    Instructions', 'LIB'),
+                               'cpu_core/cycles' => array('Cycles', 'Cycles', 'LIB'),
                                );

                        foreach($perf_stats as $string_to_match => $data)
diff --git a/pts-core/modules/turbostat.php b/pts-core/modules/turbostat.php
index a51bf1ae0..0348c6bc9 100644
--- a/pts-core/modules/turbostat.php
```

```
+++ b/pts-core/modules/turbostat.php
@@ -50,11 +50,6 @@ class turbostat extends pts_module_interface
                        echo PHP_EOL . pts_client::cli_just_bold('turbostat not found in PATH.') . PHP_EOL;
                        return pts_module::MODULE_UNLOAD;
                }
-               if(!phodevi::is_root())
-               {
-                       echo PHP_EOL . pts_client::cli_just_bold('turbostat requires root access.') . PHP_EOL;
-                       return pts_module::MODULE_UNLOAD;
-               }
                if(!is_dir($dump_dir) || !is_writable($dump_dir))
                {
                        echo PHP_EOL . pts_client::cli_just_bold('TURBOSTAT_LOG is not pointing to a directory, output will be appended to PTS test run log files.') . PHP_EOL;
```

APPENDIX — TASK PLACER SCRIPT

```
#!/usr/bin/env python3

from functools import total_ordering
from pickle import load
from subprocess import DEVNULL, Popen, PIPE, CalledProcessError
import sys
from time import time

import psutil

# TODO: Dynamically detect this
P_CORES = [0, 2, 4, 6]
E_CORES1 = [8, 9, 10, 11]
E_CORES2 = [12, 13, 14, 15]

PERF_COUNTERS = ['branches', 'branch-misses', 'L1-dcache-loads', 'L1-dcache-load-misses',
    'L1-icache-load-misses', 'cache-references', 'cache-misses',
    'cycles', 'instructions']

FEATURES = ['cpu_core/branches/', 'cpu_core/branch-misses/', 'cpu_core/L1-dcache-loads/',
    'cpu_core/L1-dcache-load-misses/', 'cpu_core/L1-icache-load-misses/',
    'cpu_core/cache-references/', 'cpu_core/cache-misses/',
    'cpu_core/topdown-retiring/', 'cpu_core/topdown-mem-bound/',
    'cpu_core/topdown-bad-spec/', 'cpu_core/topdown-fe-bound/', 'cpu_core/topdown-be-bound/',
    'cpu_core/cycles/']


with open("classifier.pkl", "rb") as f:
    clf = load(f)
with open("scaler.pkl", "rb") as f:
    scaler = load(f)

@total_ordering
class Process():
    def __init__(self, name, pid):
        self.latest_update = 0
        self.counters = {}

        self.name = name
        self.pid = pid
        try:
            self.process = psutil.Process(pid)
        except psutil.NoSuchProcess:
            print("Ghost process with PID", pid)
            self.process = None
```

```
        print("Hi, I'm new process", name)

    def add_measurement(self, counter, value, timestamp):
        self.counters[counter] = value
        self.latest_update = timestamp

    def __eq__(self, other):
        return False

    def __gt__(self, other):
        try:
            my_data = [self.counters.get(f)/self.counters.get("cpu_core/instructions/") for f in FEATURES]
            other_data = [other.counters.get(f)/other.counters.get("cpu_core/instructions/") for f in FEATURES]
            scaled = scaler.transform([my_data + other_data])
            prediction = clf.predict(scaled)
            #print(prediction)
            return prediction[0] == 1.0
        except Exception as e:
            print(e)
            return False



INTERVAL = 1000

CMD = ["perf", "stat", "-x,", "--interval-count", "1", "-I", str(INTERVAL), "-e", ",".join(PERF_COUNTERS), "-M", "tma_core_bound", "--per-thread"]

process_table = dict()
N = int(sys.argv[1])

def rebalance_affinity():
    print("Update completed, rebalancing")
    processes = list(process_table.values())
    start_time = time()
    priority_list = sorted([p for p in processes if p.process and p.process.is_running() and (p.process.cpu_percent() > 20)])
    delta_time = start_time - time()
    print("Took", delta_time, "s to sort", len(priority_list), " threads, avg = ", delta_time/len(priority_list) if len(priority_list) else 1, "s per thread")
    print([p.name for p in priority_list])
    for p in priority_list[-N:]:
        print("Promoting", p.name, "to P-core")
        if p.process.is_running():
            p.process.cpu_affinity(P_CORES)
    for p in priority_list[:-N]:
        print("Demoting", p.name, "to E-core")
        if p.process.is_running():
            p.process.cpu_affinity(E_CORES1)

```

```
# Example data line from perf
# 1.462438020,kworker/u64:6-ext4-rsv-conversion-3087065,4022580,,cycles,1467413,100.00,,
class PerfDict:
    TIMESTAMP = 0
    NAMEPID = 1
    VALUE = 2
    UNIT = 3
    NAME = 4
    RUNTIME = 5
    PERCENTAGE = 6

    LENGTH = 9

def update_table(line):
    global latest_rebalance
    # We should be receiving a CSV, so I'll be parsing it manually
    datum = line.split(',')
    #print(len(datum))
    if len(datum) != PerfDict.LENGTH:
        return
    timestamp = float(datum[PerfDict.TIMESTAMP])
    pid = int(datum[PerfDict.NAMEPID].split("-")[-1])
    #print(pid)
    if not process_table.get(pid):
        process_table[pid] = Process(datum[PerfDict.NAMEPID], pid)
    try:
        process_table[pid].add_measurement(datum[PerfDict.NAME], int(datum[PerfDict.
    VALUE]), timestamp)
    except ValueError:
        print("Error getting values for", datum[PerfDict.NAME])

while True:
    with Popen(CMD, stdout=DEVNULL, stderr=PIPE, bufsize=1, universal_newlines=True) as
    p:
        for line in p.stderr:
            update_table(line)
    rebalance_affinity()


if p.returncode != 0:
    raise CalledProcessError(p.returncode, p.args)
```