Energy Efficient Scheduling in Operating Systems

John R. Smith
UT Computer Science
University of Texas at Austin
Austin, Texas
johnrsmith@utexas.edu

Nathan Williams
UT Computer Science
University of Texas at Austin
Austin, Texas
nathanwilliams@utexas.edu

Abstract—**Modern CPU architectures support technologies such as dynamic voltage and frequency scaling as well as asymmetric multiprocessing that allows systems to use less energy in periods of lower demand. However, if the operating system fails to optimize for these technologies, their savings will not be realized. Significant changes to traditional task scheduling algorithms have been recently proposed, with most evaluating the performance demand of incoming applications and the processing capabilities of each CPU core by using static or runtime modeling. Then, logical mappings will be made to ensure the greatest performance at the lowest power draw. As system demand changes, the frequencies of individual CPU cores may be lowered and processes may be re-arranged to maintain memory and processor performance. This literature review explores the various data collection and modeling approaches and scheduling algorithm paradigms with a focus on how choices impact the nature of scheduling. Ultimately, energy-aware schedulers have seen up to 30% energy savings and have, to some extent, been implemented in modern general-purpose systems.**

Keywords—operating systems, job scheduling, asymmetric multiprocessing, memory optimization, energy efficient computing

I. Introduction

With the rise of multi-core architectures and parallel applications, processors can run at a lower frequency while achieving greater performance, leading to quadratic power savings. A recent evolution of this technology is asymmetric multiprocessing, which embeds high and low performance cores on the same package. When utilized effectively, less computationally intensive tasks can be scheduled on lower-performance cores to increase energy efficiency. Dynamic voltage and frequency scaling (DVFS) has shown significant efficiency gains by reducing system frequency during periods of low processor demand. Traditionally, DVFS policies have been applied across the entire processor. This paper presents a literature review of current strategies that apply scaling on a per-core basis to optimize efficiency on the task level of granularity.

If tasks are not scheduled appropriately on the system, cycles can be wasted waiting for dependent instructions to be processed, areas of the package will be unevenly heated, and, depending on architecture, core group-dependent caches may be unevenly used. Moreover, the gains of asymmetric processors cannot be realized if the scheduler treats each core uniformly. Subsequently, there are great energy savings to be seen when considering system demand, memory/cache usage, processor core type, and job behavior to place and reassign jobs to the most appropriate core.

Expanding on this, a DVFS policy that works in tandem with the task scheduler to adjust frequencies on a per-core basis can allow the system to ramp up energy usage in a more granular manner by scheduling tasks on cores that are currently running on the most appropriate frequency. Taking inspiration from distributed and cloud platforms, which monitor system resource usage at a large-scale, general-purpose operating systems utilize classification models to analyze system resource constraints in real time to make informed scheduling decisions. Approaches that reassign tasks to the optimal core heavily utilize context switching. Classification-based approaches attempt to calculate the ideal time-slice for a newly created task and aim to mitigate context switching to avoid cache invalidation and excessive overhead.

II. Use of Dynamic Voltange-Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) is a commonly use technique to limit the power draw of a processor based on the present workload with the goal of improving energy efficiency. The gains of reducing frequency can lead to significant savings due to “the quadratic relation between power and voltage” [1]. Generally, this is achieved by first predicting the workload, computing an optimal voltage, switching part of the processor to ‘sleep mode’ and waking the processor up on a timer interrupt [6]. The contemporary approach is to operate at the core-level of granularity to better map and scale the hardware to match software demand. Modern efforts prefer DVFS over ‘dark silicon’ (where an entire core is ‘turned off’ when it is unneeded) because “powering down cores to save energy can cause large latencies” [3]. Instead, those cores would be scaled to the lowest available frequency.

The *HFEE* framework, detailed in [7], gives a valuable overview of the general approach that energy-aware algorithms take. First, “CPU power sensors and performance counters” [7] are utilized to identify both program performance and CPU energy utilization. Second, a “program suitability score” [7] is generated. Third, programs are mapped to an optimal core. Finally, some solutions may choose to scale the frequency within the architectural constraints. Much of this literature review will identify and justify specific constraints and modifications made at each level of the HFEE.

A. Instruction-Level DVFS

Many traditional task scheduling algorithms have considered task priorities based on the amount of CPU time. However, this general approach fails to consider wider hardware utilization and application behavior. [1] analyzed the per-application trends to identify and act on changes in CPU or memory utilization. Moreover, some parallel applications have instructions that "complete their execution in a dependency chain" [3], forcing the CPU to spend "cycles in latency" as it waits for I/O operations or the completion of a dependent thread. By monitoring runtime performance metrics, the algorithm proposed in [1] "lowers the voltage and frequency of the processor" during memory-intensive workloads to avoid frequent stalling.

To classify CPU or memory-intensive workloads, the *ASTPI*, or "average time spent waiting for memory access" metric is measured and updated as an application runs. This figure is quantified through measuring multiple performance counters of the CPU. The specific metrics are the *average effective time*, which "prioritizes workloads and cores [based on their] instruction sets and computational efficiency," the *effective time*, or "the time spent in CPU for actual execution," and the *stall time*, which is "the time spent by the CPU to access memory" [1]. Performance counters are typically provided by an architecture manufacturer-provided driver; approaches like [1] may consider many platform-dependent metrics to determine measures such as *ASTPI*. Workloads are then assigned to slower and faster-frequency core clusters based on their CPU intensity, measured through the latter figures.

[1] found that operating "at the threshold voltage and frequency levels for memory-intensive work" and at the maximum frequency for CPU-intensive workloads led to the best balance of performance and efficiency. More practically, CPU package temperature is balanced when a "heavy compute-intensive workload is followed by a heavy memory-intensive workload" [1]. To achieve this, their proposed algorithm uses "average effective time for workload and core prioritization," sorting the dynamically generated scores to "maximize the utilization of high-performance cores" [1].

The algorithm provides the best optimization when the "number of processing units and workload grows." Effective workload mapping that identifies changing runtime characteristics is "scalable on increased heterogeneous workloads and CPU cores" [1]. Compared to a non-DVFS task scheduling algorithm, the program execution time increased "from 27% to 33% [while saving] 67% to 72.8% energy" [1]. Running the CPU at the maximum frequency would be expected to produce the best performance. The increased overhead of runtime monitoring, incorrect scheduling choices, and aggregately lower speed explains the marginal increase in execution time. Compared to a traditional DVFS model, execution time increased "2% to 5% [while saving] 14.5% to 20% of energy" [1] This demonstrates that there is a slight tradeoff from both the overhead of the algorithm and the more aggressive voltage scaling. However, efficiency can be greatly improved when tasks are paired to their most suitable frequency – this is a common pattern among energy-efficient scheduling schemes.

B. *Energy-Efficient Workload-Aware scheduler (EEWA)*

Oftentimes, the key to making energy-efficient or performant scheduling decisions is to identify application 'phases', or periods of similar CPU or memory demand. More general applications, such as gcc, may have periods where it accesses disk or memory frequently (when it is loading in source code or accessing shared libraries for linking). Afterwards, it will see a CPU intensive period as it compiles the code. Other applications, however, may fail to show significant patterns or reveal predictable dependencies: using a phase-identifying scheduler for these may lead to wasted energy. The researchers in [3] propose the EEWA, a scheduler for applications where the work of one thread is dependent on the completion of a separate thread and the computations follow repeatable and scalable patterns. This parallel-workload scheduling approach "tunes the frequencies of the cores according to the workload information [and] balances the workloads among cores by stealing tasks according to a preference list" [3]. EEWA assumes that changing iterations will enjoy similar runtime metrics and thus the most optimal load balancing and scheduling decisions can be made at process creation time.

EEWA pre-calculates core-frequency pairs to determine what utilization and performance level the system should apply. Given two parallel and dependent tasks T1 and T2 running on cores C1 and C2, where one finishes in half the time, traditional task scheduling "will not scale down [the frequency] after C1 finishes T1 since C1 will be actively trying to get new tasks until the application terminates" [3]. EEWA aims to "reduce energy consumption without degrading performance" in similar scenarios by scaling the frequency of C1 to half, allowing both T1 and T2 to finish at the same time. Naive schedulers may run into situations where performance and energy consumption are significantly degraded if the slower core is scaled down.

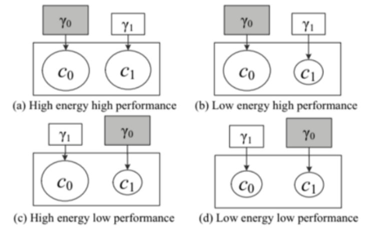
γ0
γ1
c0
c1
(a) High energy high performance
γ0
γ1
c0
c1
(b) Low energy high performance
γ1
γ0
c0
c1
(c) High energy low performance
γ1
γ0
c0
c1
(d) Low energy low performance

Fig 1. *Diagram demonstrating how poor scheduling decisions can be made for parallel and dependent tasks (from [3])*

EEWA avoids this situation and achieves energy efficiency by studying applications' computational and memory-access patterns and determining the ideal allocation of big/little cores (see Section IV for more detail). They group tasks together and create a table of frequency ranges ('task classes') that define the number of cores needed to run a task at a given frequency. At runtime, they collect workload information to update and inform the table's ranges (using performance counters). This is done by executing each task at the highest frequency and later scaling the frequency back. Finally, they use a backtracking algorithm to find an optimal frequency-core range, which runs with complexity of O(kr^2), where k is the number of task groups, and r is the number of system cores. Accordingly, "table is usually small… because cores can only run at a few fixed frequencies" [2]. This is to ensure that the application can operate without significant performance degradation while running cores at the lowest frequency possible, taking greater advantage of multi-core parallelism.

EEWA does not disregard changes in workload classes. Each CPU core is assigned a preference-ordered queue that stores each task to be scheduled. Given that this implementation uses DVFS and not dark silicon, to avoid a core wasting static energy at idle an empty task pool will pop tasks from other cores' queues according to preference levels. Priorities are based on the "rob-the-weaker-first principle," which the authors argued has shown promising results in asymmetric multiprocessing. Given that a core runs out of jobs to execute, it "is scaled down to run at the lowest frequency" [3]. This is especially prevalent in scenarios where a task runs at full speed while leaving the core underutilized.

The authors found that energy consumption reduction ranges from 8.7% to 29.8% at the cost of performance degradation ranging from 0.8% to 3.7%" [3]. They attribute the promising energy savings to "appropriately configuring core frequencies" and implementing effective workload balancing techniques. With some limitations, the algorithm scales well. When there are a few cores, "all the cores are adjusted to run at the highest frequency"; with more cores (i.e., 12), "EEWA reduces 23.8% of energy consumption" [3]. Ultimately, the EEWA scheduler demonstrates how per-thread performance data can be used to inform a core-dependent frequency. This allows the system to scale to demand and workload type in a granular fashion, diminishing the energy overhead of running a multicore processor that potentially remains mostly idle. Next, we will look at more general approaches that consider more holistic system performance metrics.

III. Resource-Aware Scheduling

Thus far, we have explored the basis for many energy-efficient algorithms. Beyond voltage scaling, it may be useful to consider a variety of system constraints such as context switches, cache misses, and memory layouts. Moreover, the way data is calculated, such as at runtime or execution time, can have a significant impact on scheduling characteristics. Schedulers that do not consider system resources (as discussed in [9]) typically result in poor turn-around time, energy consumption, and fairness depending on architecture type (addressed in Section IV). Not only will more informed scheduling improve the performance of applications, it will also lower package temperature (decreases the amount of wasted energy transistors produce) by avoiding core 'hot-spots.'

A. Feedback-Driven Priority-Based Scheduling

Context switching involves moving a process or thread from one CPU core to another. This allows it to continue execution in a more optimal environment. However, there is some overhead incurred by a context switch, including the potential for cache invalidation. Researchers in [2] consider both the context switches and cache misses of a program to decide when and where to migrate a process. They compute a per-core performance index, based on instructions per clock (IPC) and program wait time, and maintain per-core priority queues to balance system load most effectively. Notably, optimizing priorities for cache misses and context switches can lead to different scheduling behaviors, with varying tradeoffs. By finding an optimal balance of these metrics, [2] "exploits the increased performance associated with context switching more CPU intensive processes to higher frequency cores, without suffering from the performance loss associated with cache coherence" [2].

Some approaches (see [3]) allow applications to be scheduled traditionally and later 'moved' based on dynamic performance counters and Linux *cpuset*s. This algorithm, on the other hand, aims to "avoid large initial load imbalances" [2] by considering which applications are running per core and assigning new processes to cores with the lowest loads. If a core has a high load estimate, then processes in that set will be moved to cores with lower load estimates. During runtime, based on the target algorithm, a *cache miss index* or *context switch index* is calculated, with CM calculating cache misses compared to references and CS counting the total number of migrations. If a process suffers degraded performance due to long wait time or if it incurs a high CM, it may be migrated to a higher or lower performance *cpuset* queue respectively. Task priorities determine the probability of a context switch, with lower priority processes requiring lower load averages to be migrated. Memory saturation is also considered, with processes that are not CPU intensive being switched into *cpusets* with a high load (with the idea that those sets will not be actively accessing memory). Finally, to mitigate starvation, processes with a high wait time are migrated "to a *cpuset* with fewer processes" [2].

[2] found that the CM algorithm led to an increase in average cache misses due to greater process migration, while the CS algorithm successfully decreased context switches (see Table 1). By reducing "cache miss and context switching overhead" [2], the CM algorithm generated a better performance per watt. CS "created a power savings which was as high as 38 watts while simultaneously generating a performance gain of 15.34 percent" [2]. A 50 percent reduction in the average number of context switches implies the "reduction of context switching overhead generated by this algorithm is high enough to increase performance but is not so high as to stifle the context switching necessary" [2]. This algorithm represents a more holistic approach to scheduling, using many heuristics to measure wait time, performance, power consumption, and load balancing. By considering *cpuset* queues as well as process priority, this scheduler can optimize for fairness (i.e., response time) over purely energy or performance (i.e. turnaround time). Moreover, optimizing for different metrics (i.e., cache miss versus context switch) based on application type or a given system load may also lead to better performance per watt.

TABLE I. FDPBS RESULTS

FDFBS Results	Improvement over Linux Scheduler	
	Cache Miss	*Context Switch*
Performance	1%	1.2%
Power Savings	21.4%	20.8%
Execution Time per Watt	15.2%	24.6%

B. Contention Reduction – Distributed Intensity Online-Energy

Contention, as it is described in the paper by [5], is when the number of hardware components which are trying to use a particular resource exceeds the maximum bandwidth that that resource can sustain at any given time. The result is a bottleneck. [5] found that in multicore CPUs, data is often bottlenecked by a significantly slower memory unit. This is not news – discussions of the 'memory wall' go back to Wulf et al.'s 1995 paper *Hitting the Memory Wall.* According to [13], given logical threads that share computing resources, "processes scheduled to run on different threads at the same time contend for resources." Moreover, cores on the same NUMA node "share memory and I/O bandwidth" [13]. Implementations may attempt to balance the resource load evenly by "spreading the contending threads uniformly across different shared resource levels" [8] such as core cache or memory controllers. When considering asymmetric architectures, [8] argues process mapping "should take into account that threads could be potentially harmful to each other in the usage of shared resources (i.e., cache, memory)," but most leading implementations assume that a single core is executing a single thread.

[8] proposes a solution which aims to consider "CPU and memory demands of threads… co-running on the same core." They contend that a one-sided approach, which uses only a metric such as retired instructions or last-level cache misses, provides a misleadingly optimistic view of performance as they do not consider the suitability of the thread co-runner. [8] further notes that threads which have a "high/low cache miss rate do not necessarily have high/low retired instructions" and that stall cycles fail to distinguish memory access patterns for "high-performance applications" [8]. The proposed Distributed Intensity Online-Energy (*DIO-E*) algorithm uses predicted thread behavior to evaluate the case-specific energy demands of thread co-runners.

The Energy Delay factor (ED^2) is estimated on a per-application thread basis based on instructions per second (IPS) and LLCM collected via performance counters (which accounts for phase changes). Given significant variation in the energy delay product between big and little cores, it is crucial to spread "contention for shared resources… uniformly across the cores" [8] as to not cause unfairness or specific resource overloads. First, the Linux scheduler allocates threads such that they are equally balanced on the system. Then, *DIO-E* collects dynamic performance information that determines energy factors for each thread, using the Linux *perf* utility. A "user-level monitoring module" [8] collections information about stall cycles. Like other algorithms, the energy factors are sorted, and threads are mapped in reverse order based on the availability of big/little cores. After the mapping, threads are pinned to the assigned cores to "avoid excessive migrations and provide more controlled system behavior" [8].

Compared to the state-of-the-art algorithms which do not factor in energy by merely contention equally (*DIO)* or only considering memory contention (*Policy3*), *DIO-E* did not see significant gains on a *homogenous* system. For a *heterogenous* system, however, the prior algorithms inappropriately assigned memory-intensive applications to big cores. *DIO-E* allows big cores to be used more efficiently by assigning applications that stall often to little cores. On average, *DIO-E* improved execution time by 11.7% and energy consumption by 1.8% to 57% compared to the non-asymmetric algorithms.

C. Cache-aware considerations

Expanding on [8], the level of process contention is often dependent on poor cache sharing. [6] describes that when "concurrently running threads share a single second-level (L2) cache," an inappropriate co-runner can have a significant effect of miss rates and retired instructions. Unfair performance can be estimated if two co-runners have significantly different miss rates. If a scheduler gives an 'equal share' of the CPU to all co-runners, it may not consider cache behavior. L2 cache allocation details are determined by the memory controller, not the operating system, so making scheduling decisions based on that is difficult. Due to this, [6] as well as other memory-aware fairness schedulers inform decisions based on modeling and heuristics. Similar to [8], [6]'s OS scheduler extension "redistributes time" to "influence the thread's CPU latency" and account for "performance decreases due to unequal cache sharing," informing decision.

By simply allotting a cache-starved thread more CPU cycles at the expense of the cache-hoarding thread, the suffering thread's latency can be returned to a non-cache starved state. This enables "predictable forward progress [of a thread] regardless of co-runners" [6]. To ensure fairness, the increased time slice will be returned to normal once the cache contention is eliminated. The extra time allotted to the cache-starved thread is based on L2 cache miss rate and the estimated ideal CPI and number of instructions given a fair cache; the figures are collected/derived initially, with later adjustments are made at runtime.

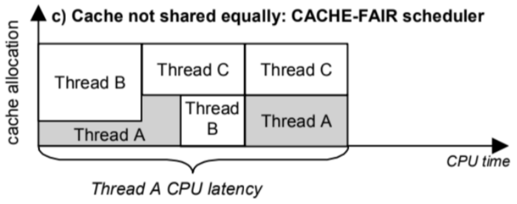

Fig 2. *Depiction of how the [6] algorithm dynamically adjusts CPU time to fairly-allocate the cache.*

D. Hardware-Based Scheduling

Modification of hardware components, such as the memory controller, can lead to increased energy efficiency. Hardware solutions often operate at a finer granularity than software solutions, and so it is worth studying how they choose to schedule their operations in a multi-threaded system.

For DRAM to service a memory request, it must first decode the physical memory address to determine what row the requested data is in, and then transfer that row into the row buffer, replacing whatever data was already there. This row replacement will occur in several DRAM chips at once, as multiple DIMMs are used in parallel to service a single request. This structure is visualized in Figure 3. [4] claims that sequential accesses to the same row are both quicker and more energy efficient than cross-row accesses since data does not need to be first transferred to the row buffer.

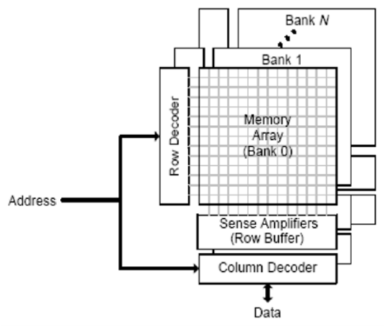

Fig. 3: *DRAM chip structure. [4]*

By prioritizing memory requests to the same row rather than handling them in a first-come first-serve approach, the system will more frequently experience the benefits of same-row accesses. However, prioritizing row accesses without considering fairness or the priority of incoming memory requests can lead to an inconsistent system. Furthermore, the performance of scheduling memory accesses is dependent on the size of the pending access buffer. A larger buffer will provide more opportunities to produce an optimal ordering of accesses, but at the cost of increasing the risk of starvation when using the naive algorithm. A smaller buffer, however, will limit the effectiveness of this naive algorithm.

Core-aware memory access scheduling, presented by [4], is a solution to the fairness problem in multi-core systems based on the principles of data locality. [4] assumes that data coming from the same core is more likely to access the same row, and therefore gives that core's memory requests a higher priority. To prevent starvation, if the number of concurrent memory requests from the priority core crosses a threshold, its priority will be transferred to another core. Cores are assigned priority in a round-robin fashion.

The result was a reduction in memory requests, latency, and execution time across all tests. Though the energy use was not measured, it can likely be concluded that this would result in energy savings as well.

IV. Asymmetric CPU Scheduling

Asymmetric multiprocessing is a recent expansion of multicore hardware design that implements cores of varying performance levels and designs. This allows software to more effectively balance and schedule tasks to conserve energy while maintaining acceptable performance. This design may involve multiple cores with "a specific microarchitecture, voltage and frequency levels, cache size and pipeline stages" [3]. Notably, manufactures implement "fast and complex cores (aka. big cores) [that] execute the serial code sections, while slow and simple cores (aka. little cores) crunch numbers in parallel" [3]. More modern techniques may add even more hierarchies and clustering. CPU-intensive workloads will benefit from higher-performance cores, whereas memory-bound tasks can spend less energy executing on slower cores as they often sit idle. [9] notes that the increased flexibility and performance granularity granted by this architectural style, "heterogeneous multi-core architectures outperform homogeneous platforms" [9]. For example, Apple's switch to the ARM big.LITTLE architecture has produced devices with long battery life and performance that can scale to user demands.

A. Core-Efficiency Task Mapping

Given the performance and subsequent energy consumption variations of asymmetric multiprocessors, it is critical for schedulers to pair processes to their most effective cores. With differing core performances, more than just the cache-aware algorithm of [6] must be considered. If a demanding application were to be scheduled on a little core, it may lead to an undesirable impact on performance. Likewise, if every task is automatically scheduled on big cores, no energy savings could be realized. Expanding on the techniques of estimating/calculating application demands and mapping them to frequency-reduced cores, [9] employs static code analysis as well as adjustments based on runtime performance counters to "map programs to the most appropriate core [types]" [9].

Due to the limitations of thermal design power (TDP) metrics and lack of required monitoring hardware for exact energy measurements, [9] devised a regression model to make predictions of current energy usage. Scheduling decisions were made in tandem with observed application execution times and performance counter data collected from the Linux API. The training dataset consisted of "fifteen pieces of hardware performance data" [9] including cycle count, instruction counters, cache accesses, TLB misses, and branch mispredictions, collected across four SPEC benchmark suites. Principally, static energy was observed from "execution time" while dynamic energy was predicted based on "retired instructions, L1-D cache accesses and L2 cache accesses" [9].

This data was utilized to identify program phases that indicated whether a frequency adjustment should occur. Whether a program will change phases can be predicted at compile time through static analysis, based on the observation that per-phase energy consumption is similar "across different invocations of the same function and loops" [9]. To identify a program phase, the LLVM compiler was used to analyze the "boundaries of function calls and loops" and to construct a "caller-callee relationship" [9]. At each call stack node, the "number of invocations and… executed instructions" [9] was counted. A phase was detected if "the total number of instructions executed" as well as "the number of invocations" each met a threshold, which is set to avoid excess "thread migration… overhead" [9]. To determine consumption behavior, the first two schedules of a program are run on the high-energy core. Afterwards, the scheduler "selects the core that has a lower energy delay product" [9] for future iterations.

The researchers found promising results with this algorithm, achieving "an average 10.20% [in energy] reduction [and] up to 35.25% reduction in the energy delay product" [9]. Ultimately, the algorithm "can achieve an average 19.81% reduction in energy delay product," with improvements increasing with the number of job iterations due to "fewer number of context switches needed" [9], whose cost is 20 µs on performance and 40 µJ on energy. However, programs that lack identifiable phases (i.e., ones that consistently have the same CPU demand or may never be memory-intensive) do not see great benefits from this algorithm. Moreover, scalability may become an issue as the number of cores increases, given the O(PN) complexity, where N is the number of cores and P is the number of processes. Ultimately, the methodology laid out in [9] demonstrates how predictions of the energy delay product upfront can be used to uncover critical patterns while reducing runtime data collection overhead; a tradeoff of program turn-around-time to improve response time.

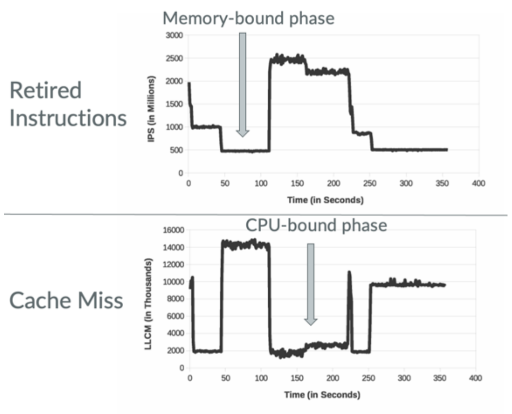

Fig. 4: *Annotated depiction of the relationship between instructions and cache misses, demonstrating identifiable periods of memory or CPU activity (from [8]*

B. *Fairness-Based Scheduling*

Creating a mapping that employs energy and performance as first-class constraints may lead to an impractical scheduler that suffers from unfairness. [7] makes the case that schedules that do not consider fairness (i.e., reducing starvation) will lead to "reduced performance predictability, a longer makespan, starvation, and QoS degradation" [7]. This is because asynchronous architectures implicitly "suffer more unfairness compared to homogeneous processors" [7], with unfairness increasing in relation to greater core performance differences. To subvert this while conserving energy, they propose a framework that uses DVFS in tandem with traditional energy-efficient scheduling techniques. Similar to the *EEWA* [3], this approach applies scaling at the cluster-level granularity, where each cluster "supports the same voltage/frequency pairs" [7], and measures the effect that scheduling decisions has on system usability.

A scheduler is considered fair "if the variation of performance degradation on a big core with highest voltage and frequency is minimal" [7]. The energy efficiency ratio (EER) indicates core suitability for a process; this is found by dividing the consumption and instructions per second of the core types. A higher EER indicates that a program would benefit from running on a big core, and an EER of 0.5 indicates starvation may occur. A completely egalitarian system would see a universal process EER - implementing this, however, would be contrary to the goals of asynchronous processing. Instead, the priority of a process is derived from scaling the EER by how long it has waited to be scheduled. Similar to [1], program EER scores are ranked and evenly scheduled based on suitability. The Instruction per Watt (IPW) ratio of big to little cores "serves as an indicator of a program's suitability for each cluster type" [4]. Their algorithm classifies processes with a higher ratio as more suitable to run on big cores.

To gauge which decisions to make to achieve high performance and fairness, "various sets of workloads are selected and executed over combinations of big and little frequency levels and the fairness value is calculated" [7]. Uniformity, which measures scheduler or frequency scaling slowdown, generally represents how varied the performance characteristics of a system are. Higher uniformity is achieved by decreasing the big core's frequency "until the two clusters' computing power become rather equal" [7]. The system can be in four states depending on uniformity and frequency; ideally, processes should be scheduled to achieve high uniformity and low frequency. Runtime scheduling scheme modifications can be made that are informed by Uniformity. For instance, in the case that few programs are running, "all remaining applications are migrated to big cluster, which improves fairness significantly" [7], especially if a lower frequency is used on the big core. The Uniformity metric allows it to evaluate how its choices are felt by the user and is ultimately most useful when considered in tandem with the metrics derived by other approaches. By defining both frequency and Uniformity thresholds, users can practically tune their system parameters to meet their workload demands.

TABLE II. ALGORITHM RESULTS

HFEE Results	**Percent Improvement by HFEE**			
	Uniformity	***EDP***	***Makespan***	***Energy***
Linux	57.6%	68%	57%	33%
Min-Fair	-3%	57%	27%	51%
DTPM	51%	61%	65%	-9%

Table II shows a pertinent summary of the results of [7]. Most notably, their algorithm generally saw collective increases in energy consumption and process uniformity compared to other leading algorithms. Individually comparing the results to best-in-class solutions saw [7] fall short; however, the authors' goal was to provide a solution optimized to asynchronous processors. A weakness of other solutions, such as the EEWA [2], was a large runtime complexity. Compared to a traditional scheduler, [7]'s overhead was empirically observed at "about 1.7%" [7]. This algorithm "is bounded by O(|P|log(|P|)" [7], where P is the number of programs, meaning that measuring Uniformity does not impose an impractical implementation complexity. We believe that modern general-purpose operating systems should follow a similar framework.

C. Energy-Aware Scheduling in Linux Today

Real-world systems may face many more constraints than just energy, fairness, and performance. The approaches that have been discussed evaluate the experimental gains factoring in energy consumption as a first-class scheduling constraint. However, some of the methods may be too application-specific or rely on empirically derived data from benchmarks. [12] reflects the complex considerations that kernel developers for widely adopted operating systems face, ultimately describing the adopted changes to the Linux deadline scheduler to "provide an acceptable QoS" while "[reducing] the energy consumption" [12].

An important constraint is satisfying the critical deadlines of real-time applications. Although users of general-purpose systems can expect runtime variation between execution due to dynamic voltage scaling, resource contention, or system load, scientific and embedded applications may fail under these implementations. Specifically, frequency level estimation based on CPU load may "risk [breaking] timing guarantees of real-time activities" [12]. But, setting the frequency to a hard maximum value "increases energy consumption" [12] due to underutilization of DVFS. The kernel-integrated solution found an energy-aware compromise within these frequency-scaling constraints. Ultimately, the Linux kernel community is most concerned with "energy efficiency rather than precise real-time execution" [12].

[12]'s solution implements the Constant Bandwidth Server (CBS) algorithm, which applies the earliest-deadline first principle to allocate per-period time slices. This can be thought of as a kind of Multi-Level Feedback Queue, where scheduling decisions are determined based on how tasks performed in a previous time slice. A task executing for more than the assigned period is penalized by being scheduled less often to mitigate starvation or the violation of real-time guarantees. An appropriate task time slice is assigned based on the current CPU speed, which is dynamically scaled based on an estimation of active CPU utilization which increases when a process wakes up and decreases after it blocks for a period of time. The GRUB-PA algorithm identifies "reclaimable bandwidth" [12] to lower CPU frequency. The execution time of tasks scales proportionally with the lowered frequency while maintaining deadlines.

The modern Linux kernel supports fractional inactive bandwidth reclaiming on a per-core basis to prevent excessive starvation of lower-priority tasks. The user can explicitly request further reclaiming through a per-task flag. Kernel developers considered two avenues for monitoring per-core demand. Scaling back the frequency based on active run queue utilization results in aggressive scaling and subsequent switching. Total run-queue utilization, on the other hand, results in less energy savings while producing better performance and fewer frequency switches. Frequency scaling and accounting operations must be performed on a per-queue basis to account for asynchronous frequency changes from different task-scheduling classes. The paper ultimately implemented these changes via minimal modifications to a DVFS governor based on the modularity of the *cpufreq* class.

There are some difficulties with this approach based on architectural differences. Principally, the currently reported frequency may be slightly inaccurate; the 'lag-time' of between receiving increased workload and taking action on frequency changes may lead to performance degradation. Moreover, some CPUs may only support limited frequency scaling states (as frequency cannot be arbitrarily scaled) or take more time for actually switching frequency, limiting the effectiveness of these algorithms.

Some studies rely on theoretical, software-based measures of energy consumption (oftentimes basing benchmark results on the figures reported from existing calculations used within the algorithm). That approach will often yield limited accuracy. [12]'s approach, however, empirically measures the consumption (albeit using only the ARM architecture) with an embedded meter. Accordingly, [12] found that their mechanism did "not introduce any unexpected deadline misses" for tasks where real-time performance could be guaranteed (i.e. when small variations in the execution time do not result in missing). This was in stark contrast compared to the *schedutil* algorithm (what the proposed algorithm replaced), which relied on the requested CPU frequency and not the actual CPU frequency and failed to compensate for frequency switch times. However, deadline misses are common if the taskset is not guaranteed to miss a deadline and when utilization is greater than 60%. The new algorithm "considerably reduces the energy consumption when the utilization… is lower than 70% while maintaining good real-time performance" [12]. Compared to the *schedutil* algorithm, the new algorithm has increased energy consumption "for utilizations larger than 70%" [12] due to the processing overhead of frequent frequency switches.

As an aside, it is important to outline the current state-of-the-art in the Linux kernel for general purpose and fair scheduling. Modular scheduling classes can be assigned on a per-task group basis for real-time scheduling, fair scheduling, or other demands. The most prevalent class, the Completely Fair Scheduler (CFS) aims to prevent starvation and mitigate resource contention through active load balancing. CFS implements a self-balancing tree to sort run-queue tasks, giving priority to the least executed tasks via a virtual runtime. A high-priority task will have its runtime accumulate more slowly, allowing it to be chosen more frequently. Load balancing is performed periodically by migrating work from busy to free cores based a "heuristic estimation on the cache and NUMA locality" [13] to determine cache usage. The first idle CPU is then chosen "to avoid unnecessarily repeated work" [13]. CFS is an incredibly complex algorithm, comprising much of the Linux scheduling code, and is not optimized for real-time tasks (hence why the scheduler is so modular). Due to its improvement in latency and context switching, however, it has been widely adopted.

D. Mapping-Based Scheduling Analysis

Although the core of these mapping algorithms is similar, each optimizes for varying system characteristics. For workloads that do not see much phase variation, [3]'s approach may be most effective. As mentioned, [8 and 9]'s approach is more valuable for systems with clear phases, where application memory accesses and CPU utilization may vary greatly during runtime. Ultimately, a combined approach, which also factors in [7 and 12]'s arguments for algorithms that equally optimize for energy, fairness, and performance, may be a lucrative venue for future research.

V. Machine-Learning Based Approaches

Many of the approaches discussed either apply complex observational-based policies [2 and 13] or utilize runtime data to make granular adjustments. Although they have shown great success, there may be benefits to collecting a wider array of data and training a classification model to make more intelligent scheduling decisions by incorporating static analysis, online/offline profiling, and existing knowledge about common applications or use patterns. As [11] commented, "most algorithms… are based on educated assumptions, and those are not exactly correct." Although the existing corpus of research doesn't encompass more sophisticated data points, promising results have been found by analyzing both existing programs [11] and per-program ELF data [10].

These models aim to classify the time slice that should be assigned, potentially using much more sophisticated data than the kernel can instantaneously collect. When a task is scheduled with a time slice that is lower than what is required for execution, early preemption "increases the number of context switches… [leading to] invalidation of caches and pipelines [and] swapping of buffers" [10]. A well-trained classification model can reveal deeper patterns in incoming programs. Time slices tailored to fit both individual applications and overall system behavior would reduce the number of context switches (avoiding the mentioned problems) necessary to ensure a fair, performant, and energy-efficient environment.

A. ELF Header-Based Analysis

[10] argues that a strong correlation between memory-intensiveness and cache misses/retired instructions is not always present. Subsequently, they utilize machine learning to predict optimal CPU time slices "by an analysis of certain static and dynamic attributes of the processes while they are being run" [10], minimizing process turn-around time (TaT). Specifically, the Decision Tree classification algorithm was used as it "approximates discrete-valued functions and is robust to noisy data and capable of learning disjunctive expressions" [10]. The underlying changes were integrated into the kernel with minimum modification. They added a field to the Linux *task_struct*, called *special_time_slice*, which allowed a process to "request a larger amount of processor time," enabling the scheduler to identify "CPU-bound processes" [10], and ultimately informing it whether to context switch a process on preemption. System calls to increase the *special_time_slice* and return a process to a normal scheduling state were also implemented.

When calculating the *time_slice*, "each process has to be checked to see if it is CPU-bound" [10]. This allows the result of the classification to be factored in when considering if a process "can be allotted more than task_timeslice ticks" [10]. The classification is collected from program ELF headers - the fields InputSize, ProgramSize, BSS, RoData, Text, and InputType were shown to be most effective. The algorithm determines an STSClass and returns this via a system call to the modified scheduler, which appropriately increases the program execution time by that value.

Data were collected from only 84 instances of five benchmark applications: the applicability of this method may be greatly expanded if a data collection apparatus were developed for traditional personal operating systems. Considering this, [1-] found that the *special_time_slice* converged on an ideal value, showing "about 60320 micro-seconds saved per second." On a 1.6Ghz single core processor, this resulted in saving "16 x 108 clocks/second = [16 x 108 x no.of pipelines x (60320)] low-level operations" [10], which may translate into energy savings given the overhead of running the model is not significant.

Benchmarking on Matrix Multiplication, this algorithm saw an average turn-around-time savings of 1464 milliseconds. The value in [10]'s approach its modularity. Applying the same framework, different types of data may be collected across different classes of applications, both at runtime and before. Further, applying different approaches such as clustering or deep learning may reveal even more significant patterns. Moreover, time slice predictions can be pipelined with voltage scaling and asymmetric multiprocessing schedulers to realize even more aggressive energy savings. However, for [10] to be considered in a general-purpose operating system, programs who have different phases, hardware usage (such as GPU bound tasks) and memory demands should be tested. Will the same decision tree model be scalable to very diverse applications?

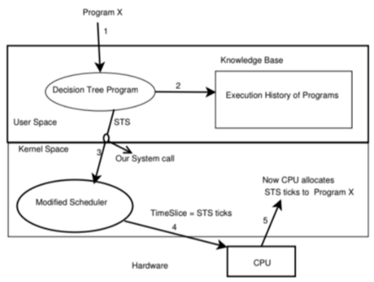

Fig 5. *Overview of [10]'s approach to implementing a modular ML-informed scheduler in Linux. Presumably, a model like this could be applied with different types of algorithms and easily switched out based on kernel flags.*

B. Database Model Analysis

The collection of data at runtime may lead to increased system overhead. Moreover, deeper patterns may be discovered from analyzing program performance and workload-nature data over a long range. [11] aimed to "minimize waiting time and process starvation by determining the optimum time quantum" through the implementation of a classification model.

A simple algorithm was implemented. Their input data was collected from the GWA-T-4 AuverGrid dataset based on "five geographically organized clusters" [11]. A linear regression model is used to predict the run time of a process based on "average turn-around time and average wait time and context switches" [11]. The scheduler keeps a priority queue, sorted based on the predicted run time. If a process does not finish within its time slice, the subsequent time slice is reduced for that process. The algorithm was evaluated based on 10,000 input processes and produced "almost half of the turn-around time" while also reducing wait time compared to leading scheduling algorithms. This better performance is due to a more optimized time slice that ultimately reduces the number of context switches necessary (they observed fewer than half the total number of switches) [11]. However, the paper did not consider fairness – a completely-optimal time slice may starve out other processes, leading to a non-uniform experience (see [7]).

C. Hybrid Approach – Replacing Existing Kernel Functions

[13] noted the weakness in the existing implementation of the CFS is that it "overlooks contention between processor cores" based on an often-false assumption that fair CPU sharing will equate to fair memory, cache, bus, or other resource sharing. [13] aimed to implement a reinforcement model that expands on CFS to "include hardware resource utilization models." Their model employs a hybrid of the data collection schemes we have discussed, utilizing "dynamic kernel tracing" as well as runtime statistical analysis to identify "potential hardware bottlenecks" [13] and perform load-balancing actions.

A multi-layer perceptron (MLP) was trained on the cache and NUMA-locality heuristic decisions that the existing CFS implementation made. To collect runtime statistics of migration decisions, both dynamic kernel tracing and kernel event monitoring was utilized and included CPU idleness, NUMA nodes, CPU load, runqueue lengths, and more. This MLP was used to replace the existing migration-advising function in CFS.

The use of floating points is generally not advised in the Linux kernel as their state is not saved on a context switch. As a result, using any in-kernel ML model that relies on floating points adds overhead - using a fixed-floating point MLP can "reduce the latency by 17%" [13].

Their MLP made the same decisions as the heuristic approach 99% of the time. However, it contributes to a 13% higher latency and 36% higher overhead, with a 0.015 higher 5-minute load than the Linux kernel, representing worse load-balancing. This is seen through the "occurrence of run-queues with around 10-22 jobs [being] higher for the system running the ML model" [13]. Despite this, there was no significant difference in the runtime of highly-process driven workloads between the two implementations. [13] believes that by implementing reinforcement-based approaches, where the predictions of the machine learning load balancer can be informed by good or bad previous predictions, implementations like theirs can far outpace the heuristic-based approaches currently employed (certainly making up for the added overhead). Furthermore, training classification models outside of the kernel can lead to low overhead in production.

We see machine learning as a promising avenue to improve both the performance and energy efficiency of both existing schedulers and may lead to significant improvements in areas such as packet and memory management and the file system. Specifically, wherever a policy's ideal implementation relies on predicting future performance, we think it would be worth investigating if developing a prediction model would lead to superior results over approaches which attempt to extrapolate patterns from statistical analysis of previous data.

VI. Conclusion

The problem of scheduling is multifaceted, with solutions in both hardware and software, informed by data computed dynamically at runtime or statically before a run.

An intriguing contrast between the two task-scheduling approaches are their attitudes towards context switching. The mapping-based approaches utilize 'task-stealing' and re-assignment to achieve the most efficiency. For example, [3] dynamically scales voltages and constantly re-evaluates program phases to allow memory-dependent periods to run at the lowest frequency. This involves some runtime overhead – not only to identify the new phase, but to potentially locate an application on a new core with a new pipeline and cache. Allowing for context switches within reasonable cache miss parameters allowed [2] to similarly migrate CPU intensive processes to high frequency cores, improving performance. Energy would undoubtedly be wasted if a program, which is currently in a memory-intensive phase, were running on a core at the highest frequency.

Cache invalidation is a key argument against algorithms that increase context switching. Especially for asymmetric cores which may contain separate L1/2 caches, data may need to be re-fetched from memory to continue calculations. This ultimately leads to "invalidation of caches and pipelines" and "increases turnaround time" according to [8], who saw convincing performance improvements by letting programs stay on the processor for longer (although the fairness of this scheduler may be questioned). [8] and other approaches have shared similar concerns. They worried their *DIO-E* algorithm, which produces a "higher number of thread migrations" might introduce overhead and "make the workloads take longer when dynamically mapped" [8]. In fact, many of the algorithms have shown marginal decreases in performance – yet, this comes at the benefit of increased energy efficiency.

VII. References

[1] N. Kumar and D. P. Vidyarthi, "A novel energy-efficient scheduling model for multi-core systems," *Cluster Computing,* pp. 643-666, 2020.

[2] A. K. Datta and R. Patel, "CPU Scheduling for Power/Energy Management on Multicore Processors Using Cache Miss and Context Switch Data," in *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS,*, 2014.

[3] Q. Chen, L. Zheng, M. Guo and Z. Huang, "EEWA: Energy-Efficient Workload-Aware Task Scheduling in Multi-core Architectures," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, Phoenix, AZ, 2014.

[4] Z. Fang, X.-H. Sun and Y. Chen, "Core-aware memory access scheduling schemes," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, Italy, 2009.

[5] A. Merkel and F. Bellosa, "Memory-aware Scheduling for Energy Efficiency on Multicore Processors," University of Karlsruhe, 2008.

[6] A. Fedorova, M. Seltzer and M. D. Smith, "Cache-Fair Thread Scheduling for Multicore Processors," Harvard Computer Science Group Technical Report TR-17-06, 2006.

[7] B. Salami, H. Noori and M. Naghibzadeh, "Fairness-Aware Energy Efficient Scheduling on Heterogeneous Multi-Core Processors," in *IEEE Transactions on Computers*, 2021.

[8] R. Nishtala, D. Mossé and V. Petrucci, "Energy-aware thread co-location in heterogeneous multicore processors," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, Montreal, QC, 2013.

[9] J. Cong and B. Yuan, "Energy-Efficient Scheduling on Heterogeneous Multi-Core Architectures," in *ISLPED '12: Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, 2021.

[10] A. Negi and K. P. Kumar, "Applying Machine Learning Techniques to Improve Linux Process Scheduling," in *TENCON 2005 - 2005 IEEE Region 10 Conference*, Melbourne, Australia, 2005.

[11] M. A. Moni, M. Niloy, A. H. Chowdhury, F. J. Khan, F.-U.-A. Juboraj and A. Chakrabarty, "Comparative Analysis of Process Scheduling," in *2022 25th International Conference on Computer and Information Technology (ICCIT)*, Cox's Bazar, Bangladesh, 2022.

[12] C. Scordino, L. Abeni and J. Lelli, "Real-time and energy efficiency in Linux: theory and practice," *ACM SIGAPP Applied Computing Review,* vol. 18, no. 4, pp. 18-30, 2019.

[13] J. Chen, S. S. Banerjee, Z. T. Kalbarczyk and R. K. Iyer, "Machine Learning for Load Balancing in the Linux Kernel," *APSys '20: Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems,* Vols. 67-74, 4 August 2020.