Energy-efficient Query Processing in Web Search Engines

Matteo Catena and Nicola Tonellotto

Abstract—Web search engines are composed by thousands of query processing nodes, i.e., servers dedicated to process user queries. Such many servers consume a significant amount of energy, mostly accountable to their CPUs, but they are necessary to ensure low latencies, since users expect sub-second response times (e.g., 500 ms). However, users can hardly notice response times that are faster than their expectations. Hence, we propose the Predictive Energy Saving Online Scheduling Algorithm (PESOS) to select the most appropriate CPU frequency to process a query on a per-core basis. PESOS aims at process queries by their deadlines, and leverage high-level scheduling information to reduce the CPU energy consumption of a query processing node. PESOS bases its decision on query efficiency predictors, estimating the processing volume and processing time of a query. We experimentally evaluate PESOS upon the TREC ClueWeb09B collection and the MSN2006 query log. Results show that PESOS can reduce the CPU energy consumption of a query processing node up to ∼48% compared to a system running at maximum CPU core frequency. PESOS outperforms also the best state-of-the-art competitor with a ∼20% energy saving, while the competitor requires a fine parameter tuning and it may incurs in uncontrollable latency violations.

Index Terms—Energy consumption, CPU Dynamic Voltage and Frequency Scaling, Web search engines.

1 Introduction

Web search engines continuously crawl and index an immense number of Web pages to return fresh and relevant results to the users' queries. Users' queries are processed by query processing nodes, i.e., physical servers dedicated to this task. Web search engines are typically composed by thousands of these nodes, hosted in large datacenters which also include infrastructures for telecommunication, thermal cooling, fire suppression, power supply, etc [1]. This complex infrastructure is necessary to have low tail latencies (e.g., 95-th percentile) to guarantee that most users will receive results in sub-second times (e.g., 500 ms), in line with their expectations [2]. At the same time, such many servers consume a significant amount of energy, hindering the profitability of the search engines and raising environmental concerns. In fact, datacenters can consume tens of megawatts of electric power [1] and the related expenditure can exceed the original investment cost for a datacenter [3]. Because of their energy consumption, datacenters are responsible for the 14% of the ICT sector carbon dioxide emissions [4], which are the main cause of global warming. For this reason, governments are promoting codes of conduct and best practices [5], [6] to reduce the environmental impact of datacenters.

Since energy consumption has an important role on the profitability and environmental impact of Web search engines, improving their energy efficiency is an important aspect. Noticeably, users can hardly notice response times that are faster than their expectations [2]. Therefore, to reduce energy consumption, Web search engines should answer queries no faster than user expectations. In this work, we focus on reducing the energy consumption of servers' CPUs, which are the most energy consuming components in search systems [1]. To this end, Dynamic Frequency and Voltage Scaling (DVFS) technologies [7] can be exploited. DVFS technologies allow to vary the frequency and voltage of the CPU cores of a server, trading off performance (i.e., longer response times) for lower energy consumptions. Several power management policies leverage DVFS technologies to scale the frequency of CPU cores accordingly to their utilization [8], [9]. However, core utilization-based policies have no mean to impose a required tail latency on a query processing node. As a result, the query processing node can consume more energy than necessary in providing query results faster than required, with no benefit for the users.

In this work we propose the Predictive Energy Saving Online Scheduling algorithm (PESOS), which considers the tail latency requirement of queries as an explicit parameter. Via the DVFS technology, PESOS selects the most appropriate CPU frequency to process a query on a per-core basis, so that the CPU energy consumption is reduced while respecting a required tail latency. The algorithm bases its decision on *query efficiency predictors* rather than core utilization. Query efficiency predictors are techniques to estimate the processing time of a query before its processing. They have been proposed to improve the performance of a search engine, for instance to take decision about query scheduling [10] or query processing parallelization [11], [12]. However, to the best of our knowledge, query efficiency predictor have not been considered for reducing the energy consumption of query processing nodes.

We build upon the approach described in [10] and propose two novel query efficiency predictor techniques: one to estimate the number of postings that must be scored to process a query, and one to estimate the response time of a query under a particular core frequency given the number of postings to score. PESOS exploits these two predictors to determine which is the lowest possible core frequency that can be used to pro-

• *M. Catena and N. Tonellotto are with the Information Science and Technologies Institute "A. Faedo" of the National Research Council of Italy, Pisa, Italy. M. Catena is also with the Gran Sasso Science Institute, L'Aquila, Italy.*
E-mails: m.catena@isti.cnr.it, n.tonellotto@isti.cnr.it.

Manuscript received April 19, 2005; revised August 26, 2015.

cess a query, so that the CPU energy consumption is reduced while satisfying the required tail latency. As predictors can be inaccurate, in this work we also propose and investigate a way to compensate prediction errors using the root mean square error of the predictors.

We experimentally evaluate PESOS upon the TREC ClueWeb09 corpus and the query stream from the MSN2006 query log. We compare the performance of our approach with those of three baselines: `perf` [8], which always uses the maximum CPU core frequency, `power` [8], which throttles CPU core frequencies according to the core utilizations, and `cons` [13], which performs frequency throttling according to the query server utilization. PESOS, with predictors correction, is able to meet the tail latency requirements while reducing the CPU energy consumption from ∼24% up to ∼44% with respect to `perf` and up to ∼20% with respect to `cons`, which however incurs in uncontrollable latency violations. Moreover, the experiments show that energy consumption can be further reduced by PESOS when prediction correction is not used, but with higher tail latencies.

The rest of the paper is structured as follows: Section 2 provides background information about the energy consumption of Web search engine datacenters, the query processing activity, and the query efficiency predictors. Section 3 formulates the problem of minimizing the energy consumption of a query processing node while maximizing the number of queries which meet their deadlines. Section 4 illustrates our proposed solution to the problem, describes our query efficiency predictors, and the PESOS algorithm. Section 5 illustrates our experimental setup while Section 6 analyzes the obtained results. Related works are discussed in Section 7. Finally, the paper concludes in Section 8.

2 Background

In this section we will discuss the energy-related issues incurred by Web search engines (Sec. 2.1). Then, we will explain how query processing works and some techniques to reduce query response times (Sec. 2.2). Finally, we will discuss about *query efficiency predictors*, which we exploit to reduce the energy consumption of a Web search engine while maintaining low tail latencies (Sec. 2.3).

2.1 Web search engine and energy consumption

In the past, a large part of a datacenter energy consumption was accounted to inefficiencies in its cooling and power supply systems. However, Barroso et al. [1] report that modern datacenters have largely reduced the energy wastage of those infrastructures, leaving little room for further improvement. On the contrary, opportunities exist to reduce the energy consumption of the servers hosted in a datacenter. In particular, our work focuses on the CPU power management of query processing nodes, since the CPUs dominate the energy consumption of physical servers dedicated to search tasks. In fact, CPUs can use up to 66% of the whole energy consumed by a query processing node at peak utilization [1].

Modern CPUs usually expose two energy saving mechanism, namely *C-states* and *P-states*. C-states represent CPU cores idle states and they are typically managed by the operating system [14]. C0 is the operative state in which a CPU core can perform computing tasks. When idle periods occur, i.e., when there are no computing tasks to perform, the core can enter one of the other deeper C-states and become inoperative. However, Web search engines process a large and continuous stream of queries. As a result, query processing nodes are rarely inactive and experience particularly short idle times. Consequently, there are little opportunities to exploit deep C-states, reducing the energy savings provided by the C-states in a Web search engine system [15], [16].

When a CPU core is in the active C0 state, it can operate at different frequencies (e.g., 800 MHz, 1.6 GHz, 2.1 GHz, ...). This is possible thanks to the Dynamic Frequency and Voltage Scaling (DVFS) technology [7] which permits to adjust the frequency and voltage of a core to vary its performance and power consumption. In fact, higher core frequencies mean faster computations but higher power consumption. Vice versa, lower frequencies lead to slower computations and reduced power consumption. The various configurations of voltage and frequency available to the CPU cores are mapped to different P-states, and are managed by the operating system. For instance, the `intel_pstate` driver [8] controls the P-states on Linux systems[1] and can operate accordingly to two different policies, namely `perf` and `power`. The `perf` policy simply uses the highest frequency to process computing tasks. Instead, `power` selects the frequency for a core according to its utilization. When a core is highly utilized, `power` selects an high frequency. Conversely, it will select a lower frequency when the core is lowly utilized. However, Lo et. al [15] argue that core utilization is a poor choice for managing the cores frequencies of query processing nodes. In fact, the authors report an increase of query response times when core utilization-based policies are used in a Web search engine. For such reason, Catena et al. [13] propose to control the frequency of CPU cores based on the utilization of the query processing node rather than on the utilization of the cores. The utilization of a node is computed as the ratio between the query arrival rate and service rate. Then, they propose the `cons` policy which throttles the frequency of the CPU cores when the utilization of the node is above or below certain thresholds (e.g., 80% and 20%, respectively). The frequency is selected so to produce a desirable utilization level (e.g., 70%). Similarly, in our work we control the CPU cores frequencies of a query processing node using information related to the query processing activity rather than to the CPU cores utilization (see Sec. 4). To this end, we build our approach on top of the `acpi_cpufreq` driver [9]. This driver allows applications to directly manage the CPU cores frequency, instead of relying on the operative systems.

2.2 Query processing and dynamic pruning

Web search engines continuously crawl a large amount of Web pages. This collection of documents is then indexed to produce an *inverted index* [17]. The inverted index is a data structure that maps each term in the document collection to a posting list, i.e., a list of postings which indicates the occurrence of a term in a document. A posting contains at least the identifier (i.e., a natural number) of the document where the term appears and its term frequency, i.e., the number of occurrences of the term in that particular document. The inverted index is

1. `intel_pstate` is currently the default driver on Ubuntu distributions

usually compressed [18] and kept in main memory to increase the performance of the search engine [19].

When a query is submitted to a Web search engine, it is dispatched to a query processing node. This retrieves a ranked list of documents that are relevant for the query, i.e., the top K documents relevant to a user query, sorted in decreasing order of relevance score (e.g., by using the popular BM25 weighting model [20]). To generate the top K results list, the processing node exhaustively traverses all the posting lists relative to the query terms. This is computationally expensive, since the inverted index can easily measure tens of gigabytes, so *dynamic pruning* techniques are adopted [21], [22]. Such techniques avoid to evaluate irrelevant documents, skipping over portions of the posting lists. This reduces the response time as the systems avoid to access and decompress portion of the inverted index. At the same time, these dynamic pruning techniques are *safe-up-to-K*, i.e., they produce the same top K results list returned by an exhaustive traversal of the posting lists. For such reasons, in this work we apply dynamic pruning strategies to the processing of queries.

2.3 Query efficiency predictors

Query efficiency predictors (QEPs) are techniques that estimate the execution time of a query before it is actually processed. Knowing in advance the execution time of queries permits to improve the performance of a search engine. Most QEPs exploit the characteristics of the query and the inverted index to pre-compute features to be exploited to estimate the query processing times. For instance, Macdonald et al. [10] propose to use term-based features (e.g., the inverse document frequency of the term, its maximum relevance score among others) to predict the execution time of a query. They exploit their QEPs to implement on-line algorithms to schedule queries across processing node, in order to reduce the average query waiting and completion times. The works [11], [12], instead, address the problem to whether parallelize or not the processing of a query. In fact, parallel processing can reduce the execution time of long-running queries but provides limited benefits when dealing with short-running ones. Both the works propose QEPs to detect long-running queries. The processing of the query is parallelized only if their QEPs detect the query as a long-running one. Rather then combining term-based features, Wu et al. [23] propose to analytically model the query processing stages and to use such model to predict the execution time of queries.

In our work, we modify the QEPs described in [10] to develop our algorithm for reducing the energy consumption of a processing node while maintaining low tail latencies.

3 Problem Formulation

In the following, we introduce the operative scenario of a query processing node (Sec. 3.1), we formalize the general minimum-energy scheduling problem and we shortly present the state-of-the-art algorithm to solve it offline (Sec. 3.2), and we discuss the issues of this offline algorithm in our scenario (Sec. 3.3).

3.1 Operative scenario

A *query processing node* is a physical server composed by several multi-core processors/CPUs with a shared memory which holds the inverted index. The inverted index can be partitioned into *shards* and distributed across multiple query processing nodes. In this work, we focus on reducing the CPU energy consumption of single query processing nodes, independently of the adopted partition strategy. In the following, we assume that each query processing node holds an identical *replica* of the inverted index [24].

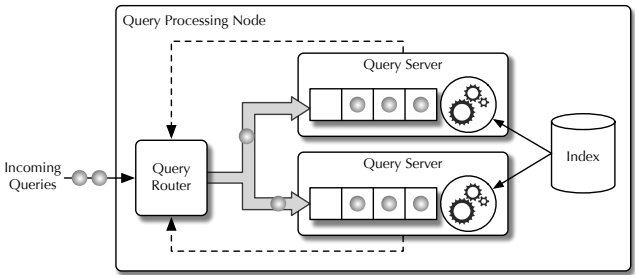

Fig. 1. The architecture of a query processing node.

A *query server* process is executed on top of each of the CPU core of the processing node (see Figure 1). All query servers access a shared inverted index held in main memory to process queries. Each query server manages a queue, where the incoming queries are stored. The first query in the queue is processed as soon as the corresponding CPU core is idle. The queued queries are processed following the *first-come first-served* policy. The number of queries in a query server's queue represents the server load. Queries arrive to the processing node as a stream $S = \{q_1, \ldots, q_n\}$. When a query reaches the processing node it is dispatched to a query server by a *query router*. The query router dispatches an incoming query to the least loaded query server, i.e., to the server with the smallest number of enqueued queries. Alternatively, the query processing node could have a single query queue and dispatch queries from the queue to idle query servers. In this work, we use a queue for each query servers since a single queue will not permit to take local decisions about the CPU core frequency to use for the relative query server. A similar queue-per-core architecture is assumed in [25], to schedule jobs across CPU cores to minimize the CPU energy consumption, and in [10] to schedule queries across different query servers.

A query $q_i \in S$ is characterized by its arrival time a_i, when it "enters" the processing node at the query broker, and its completion time $c_i > a_i$, when it "leaves" the processing node after being processed by a query server. The query processing node is required to process queries with a tail latency of τ ms (e.g., 500 ms). Therefore, we impose that each query q_i must be processed within τ time units from its arrival time, i.e., it has an absolute deadline $d_i = a_i + \tau$. If we assume negligible the time required by the query broker to dispatch the query, the completion time c_i of q_i is the sum of its arrival time, the time the query spent in the queue and its processing time. A query misses its deadline, i.e., $c_i > d_i$, if it spends more than τ time units in queue and being processed. In fact, a query may have less than τ time units to be processed. At time t, the *time budget* $b_i(t)$ of query q_i indicates how much time remains before q_i misses its deadline. $b_i(t)$ is the difference between its deadline and the time it is spending in the queue, i.e. $b_i(t) = d_i - (t - a_i)$. When a query exceeds its time budget, the query processing node has two possible choices: 1) to early

terminate the query, returning an incomplete list of results, or 2) to finish processing the query, delaying the processing of other request, but returning a complete list of results. In this work, we focus on the second option which does not degrade the quality of the search results. We do not consider here the time necessary to send the results to the users, as it involves network latencies which do not depend on the search engine.

As seen in Section 2.1, a query server can process queries at different speeds, depending to the CPU core operational frequency. To reduce deadline violations, CPUs cores can operate at their maximum processing frequency. In fact, high frequencies lead to faster computations at the price of high power consumption. Conversely, lower frequencies mean slower computations, with lower power consumptions.

Since the number of queries received by a query processing node along a day varies, we envision the possibility to dynamically change the CPU core frequencies of query servers to the number of queries received per time unit. Our goal is to maximize the number of queries that are processed within their deadline, in order to obtain a tail latency close to τ ms. At the same time, we want to minimize the energy consumption of the processing node. In other words, for each query q_i we need to select the most appropriate frequency $f \in F$ for the CPU core associated to the server processing q_i.

3.2 The minimum-energy scheduling problem

Consider the following scenario, where a single-core CPU must execute a set $J = \{J_1, \ldots, J_n\}$ of generic computing jobs rather than queries. Jobs must be executed over a time interval $[t_0, t_1]$. Each job J_i has an arrival time a_i and an arbitrary deadline d_i which are known a priori. Moreover, each job J_i has a processing volume v_i, i.e., how much work it requires from the CPU, and jobs can be preempted. The CPU can operate at *any* processing speed $s \in \mathbb{R}^+$ (in time units per unit of work) and its power consumption is a convex function of the processing speed, e.g., $P(s) = s^\gamma$ with $\gamma > 1$ [7].

Jobs in J must be scheduled on the CPU. A *schedule* is a pair of functions $S = (\psi, \phi)$ denoting, respectively, the processing speed and the job in execution, both at time t. A schedule is *feasible* if each job in J is completed within its deadline. The *minimum-energy scheduling problem* (MESP) aims at finding a feasible schedule such that the total energy consumption is minimized, i.e.,

$$\arg\min_{S=(\psi,\phi)} E(S) = \int_{t_0}^{t_1} P\big(\psi(t)\big)dt \tag{1}$$

The MESP is similar to an offline version of our problem, where jobs, corresponding to queries, are preemptable, and processor speeds can assume any positive value.

The YDS algorithm [26] solves the MESP in polynomial time. Consider an interval $I = [z, z'] \subseteq [t_0, t_1]$ and the set of jobs in that interval $J_I = \{J_i \in J : [a_i, d_i] \subseteq I\}$. The *intensity* $g(I)$ of interval I is the ratio between the amount of work required by the jobs in J_I and the length of the interval

$$g(I) = \frac{1}{z - z'} \sum_{J_i \in J_I} v_i \tag{2}$$

A feasible schedule must use a processing speed $s \geq g(I)$ during the interval I, or jobs will not meet their deadlines if $s < g(I)$. Moreover, $P(g(I))$ is the lowest possible power consumption on the interval I, since P is a convex function.

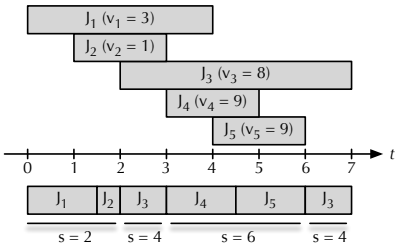

Fig. 2. An example of YDS scheduling: (top) input jobs, (bottom) resulting optimal schedule with CPU speeds s.

Algorithm 1 illustrates the YDS algorithm, that optimally solves the MESP in $O(n^3)$ [26], [27]. YDS works by analyzing each possible time interval I included in $[t_0, t_1]$. Then, it finds the *critical interval* I^* that maximizes $g(I)$. YDS schedules the jobs in J_{I^*} using the *earliest deadline first* (EDF) policy [28] and processing speed $g(I^*)$. Then, if not preempted, the jobs in J_{I^*} will terminate in $r_i = v_i \cdot g(I^*)$ time units since the beginning of their execution. Jobs in J_{I^*} are then removed from J. The interval I^* as well is removed from $[t_0, t_1]$, i.e., it cannot be used to schedule jobs other than those in J_{I^*}. For this reason, YDS updates the arrival times and deadlines of the remaining jobs to be outside I^*. Finally, YDS repeatedly finds a new critical interval for the remaining jobs, until all jobs are eventually scheduled. Note that the MESP always admit a feasible schedule, since arbitrary large amounts of work can be performed in infinitesimal time when $s \to \infty$.

Algorithm 1: The YDS algorithm

```
Data: A set of jobs J = {j_1, ..., j_n} to schedule in [t_0, t_1]
Result: A feasible schedule S for J minimizing E(S)
OYDS(J):
1   ψ ← {}
2   φ ← {}
3   while J ≠ {} do
4       Identify I* = [z, z'] and compute g(I*)
5       Set processor speed to g(I*) for jobs in J_I* in ψ
6       Schedule jobs in J_I* according to EDF in φ
7       Remove I* from [t_0, t_1]
8       Remove J_I* from J
9       foreach J_i ∈ J do
10          if a_i ∈ I* then
11              a_i ← z'                    // Update arrival times
12          if d_i ∈ I* then
13              d_i ← z                     // Update deadlines
14  return S = (ψ, φ)
```

Figure 2 shows an example for YDS. Input jobs are illustrated in the upper part of the picture. The left end of a box indicates the arrival time of the job, while the right end indicates its deadline. Processing volumes for the jobs are reported inside the relative boxes. The bottom part of the picture illustrates the optimal solution provided by YDS. The picture shows the order in which the jobs are scheduled, their start and end time, and the processing speeds s used for each job. Note that J_3 is executed over two different time intervals, as it is preempted to schedule J_4 and J_5, which have an higher joint intensity.

3.3 Issues with YDS

YDS finds an optimal solution for the MESP, but poses various issues that make difficult to use it in a search engine to reduce its energy consumption:

1) YDS is an offline algorithm to schedule generic computing jobs and cannot be used to schedule online queries. In fact, YDS input is the set of jobs to be scheduled in a interval, with their arrival times and deadlines, that must be known a priori. In contrast, query arrival times are not known until query arrives. Moreover, YDS relies on EDF, which contemplates job preemption. Context switch and cache flushing cause time overheads with non-negligible impacts on the query processing time. Therefore, preemption is unacceptable for search engines.
2) YDS requires to know in advance the processing volumes of jobs. Conversely, we do not know how much work a query will require before its completion.
3) YDS schedules job using processing speeds (defined as units of work per time unit). The speed value is continuous and unbounded (i.e., the speed can be indefinitely large). However, the frequencies available to CPU cores are generally discrete and bounded.

For such reasons, in the following Section we modify YDS in order to exploit it in a search engine.

4 Problem Solution

YDS has several issues that make unfeasible to use it in a search engine. In the following, we discuss:

1) an heuristic based on YDS which works in online scenarios without job preemption (Sec. 4.1),
2) a methodology to estimate the processing volume of a query (Sec. 4.2),
3) an algorithm to translate processing speeds into CPU core frequencies (Sec. 4.3).

Eventually, we introduce and discuss our approach to select the most appropriate CPU core frequency to process a query in a search engine (Sec. 4.4).

4.1 On-line scheduling without preemption

Online YDS[2] (OYDS) is an heuristic for the online version of the MESP, proposed in [26]. In an online scenario, we are not given a set of jobs over a fixed time interval, but the set of jobs that must be processed by the CPU changes over time. Every time $\hat{t}$ a new job arrives, OYDS considers the newly arrived job and all the jobs still to be (completely) processed, and computes an optimal solution using YDS for this set of jobs, assuming that all such jobs have the same arrival time $\hat{t}$. As YDS, OYDS guarantees that each job will be terminated by its deadline. In fact, it can schedule any processing volume by simply using an arbitrarily large processing speed s. On the other hand, its energy consumption can be sub-optimal.

While OYDS is an heuristics for the online version of the MESP, it still schedules jobs using the EDF policy which contemplates job preemption. However, in our operative scenario we deal with queries rather than generic computing jobs. Preemption is unacceptable for search engines and a query cannot be preempted once its processing has started. Since all queries must be processed within the same relative deadline τ, for any two queries q_h and q_k, such that $a_k > a_h$, we have $d_k > d_h$, i.e., later queries have later deadlines. As a consequence, EDF will always schedule firstly the earliest query, without any preemption. This means that, under these conditions, EDF coincides with the *first-in first-out* (FIFO) scheduling policy. We will use OYDS as a base for build our frequency selection algorithm, described in Section 4.4. In the remaining of this work, then, we will stop discussing about generic computing jobs but we will focus on the processing of search engine queries.

2. In the original paper, OYDS is called Optimal Available (OA). In this work, we will use OYDS for the sake of clarity.

4.2 Predicting processing volumes

The OYDS heuristic must know the processing volumes of the queries to schedule. For this purpose, we propose to use the number of scored posting during the processing of query. Indeed, for queries with the same number of terms, the number of scored postings correlates with their processing times [10]. If exhaustive processing is performed, it is possible to know a priori the number of scored postings, which is equal to the sum of the posting lists lengths of the query terms. However, when dynamic pruning is applied we do not know in advance how many postings will be scored, since portions of the posting lists could be skipped. Then, we need a way to predict the number of scored posting for a query.

We use the query efficiency predictors (QEPs) described in [10] but we modify them to predict the number of scored postings for a query. This means that we learn a set Π of linear functions $\pi_x(q)$ that, given a query q with x query terms, estimate the number of scored postings.

We note that OYDS requires exact query processing volumes. If the reported processing volumes are less than the actual ones, the algorithm does not guarantee that all the queries deadlines will be meet. QEPs are not precise, but they give only an estimate on the number of scored postings. For this reason, we add an offline validation phase after the QEPs training. During the validation, we use the regressors in Π to predict the number of scored posting for a validation set of pre-processed queries. Then, we record the root mean squared error (RMSE) for the predictions. In the online query processing, we use the RMSE ρ_x of predictor π_x to compensate its errors, by adding ρ_x to the predicted number of scored postings. In other words, our modified QEPs $\widetilde{\pi}_x(q)$ will be

$$\widetilde{\pi}_x(q) = \pi_x(q) + \rho_x. \tag{3}$$

In this way, we will likely over-estimate the processing volume of some queries, requiring higher processing speeds at the cost of higher energy consumptions. However, we will miss less deadlines, as we reduce the number of queries for which we predict fewer scored postings lower than the actual ones.

4.3 Translating processing speeds into CPU frequencies

CPU cores can operate at frequencies $f \in F$, where F is a discrete set of available frequencies (measured in Hz). Nevertheless, OYDS assigns processing speeds (seconds per unit of work) to queries. Therefore, we need to map processing speeds to CPU core frequencies. To do so, for each frequency f we train a single-variable linear predictor $\sigma_x^f(q)$, which forecasts the processing time of a query q composed by x

terms at frequency f through the estimated number of its scored postings:

$$\sigma_x^f(q) = \alpha_x^f \widetilde{\pi}_x(q) + \beta_x^f, \tag{4}$$

where α_x^f and β_x^f are the coefficients learned by the regressors. Thus, we learn offline a new set Σ of single-variable linear regressors σ_x^f, one for each frequency f. Once again, we add a validation phase after the training to build Σ, similarly to approach described in Section 4.2. We compensate a predictor error adding its RMSE (ρ_x^f) computed over the validation queries to the actual prediction, i.e.,

$$\widetilde{\sigma}_x^f(q) = \sigma_x^f(q) + \rho_x^f. \tag{5}$$

We can use Σ to translate processing speeds to CPU core frequencies, as shown in Algorithm 2. When a query q_i is associated to a processing speed s by OYDS, we compute its required processing time r_i by multiplying the predicted number of scored postings $\widetilde{\pi}_x(q_i)$ by s. Then, we check each regressor $\widetilde{\sigma}_x^f(q_i)$ in Π' in ascending order of frequency f. If the expected query processing time at frequency f is less than r_i, we use frequency f to process q_i. If we are not able to find a suitable frequency f, we use the maximum available frequency.

Algorithm 2: The CPU core frequency selection algorithm

```
Data: A query q_i composed by x terms, and
      the processing speed s assigned by OYDS to q_i
Result: The core frequency f to use to process q_i
SelectFrequency(q_i, s):
1   r_i ← π̃_x(q_i) · s
2   foreach regressor σ̃_x^f in Σ, in ascending order of f do
3       r_i^f ← σ̃_x^f(q_i)
4       if r_i^f ≤ r_i then
5           return f
6   return max_{f∈F}{f}
```

As shown in Algorithm 2, a suitable frequency f among the frequencies of the CPU cores for a query q_i does not always exists. For example, this happens when the query server is overloaded with queries to process. However, we can ignore this scenario by assuming that a query processing node has a computing capacity that, at maximum frequency, is sufficient to process its peak query volume. Moreover, a suitable frequency for a query q_i cannot be found if, at time t, q_i requires a processing time that is greater than its time budget $b_i(t)$. In such cases, we use the maximum CPU core frequency to minimize that query processing time.

4.4 Frequency selection algorithm for search engines

In this section, we describe PESOS (Predictive Energy Saving Online Scheduling). PESOS is an algorithm to select the most appropriate frequency to process a query in a search engine. Our algorithm is based on OYDS, but exploits predictors which can be inaccurate. Because of wrong predictions (see Sec. 4.2 and Sec. 4.3), some queries will miss their deadline no matter the selected CPU core frequency. Yet, this can happen because either queries have low time budgets or they require too much processing time. We call these *late* queries. Conversely, we call *on time* queries those that will be completely processed by their deadline.

Given a query q_i with deadline d_i and completion time c_i, we define its *tardiness* as $T_i = \max\{0, d_i - c_i\}$. As such, an on time query will have 0 tardiness, while a late query will have a tardiness given by the amount of time a query requires to be completed exceeding its deadline. While missing a query deadline is always undesirable, low tardiness values are still better than higher ones. Therefore, we aim at minimizing the tardiness of late queries, by reducing the time budget of on time queries. Given a queue of queries Q sorted by arrival time, we compute the total tardiness of the late queries in Q when all queries are processed at maximum frequency. Then we compute the *shared tardiness* $H(Q)$ of the on time queries in Q by dividing the total tardiness by the number of on time queries in Q, and we reduce the on time queries' deadlines by $H(Q)$. Hence, on time queries are required to finish their processing earlier, but this will leave more time to late queries and reduce their actual tardiness. Algorithm 3 recaps the steps to compute the shared tardiness $H(Q)$.

Algorithm 3: The algorithm to compute the shared tardiness of a query queue

```
Data: The query queue Q and the current time t
Result: The shared tardiness quantity H(Q)
ComputeSharedTardiness(Q, t):
1   T ← 0                                   // Total tardiness
2   n ← 0                                   // On time queries
3   f̄ ← max_{f∈F}{f}                        // Maximum frequency
4   foreach query q_i in Q do
5       b_i ← τ − (t − a_i)                 // Remaining budget
6       r_i^f̄ ← σ̃_x^f̄(q_i)                  // Max processing speed
7       if r_i^f̄ > b_i then
8           T ← T + (r_i^f̄ − b_i)           // Late query
9       else
10          n ← n + 1                       // On time query
11  return T/n
```

Algorithm 4 describes how PESOS sets the most appropriate core frequency to process a query. The algorithm works as follow. Assume q_1 is the first query in the query queue Q of a query server. At time t, query q_1 begins being processed. Initially, we check if q_1 is going to meet its own deadline. If the query is late, we set the core at its maximum frequency. Otherwise, we compute the shared tardiness $H(Q)$ of the queued queries and we change the deadlines of all the queries in Q accordingly, i.e., for all q_i in Q, we set $\widetilde{d}_i = d_i - H(Q)$. In doing so, we should just reduce the time budgets of the on time queries to leave more time to late queries. In fact, reducing the time budget of late queries has no effect since late queries will be in any case processed at maximum core frequency. Nevertheless, we reduce all the time budget by $H(Q)$ such that, for each couple of queries $q_j, q_k \in Q$, if $d_j \geq d_k$ then $\widetilde{d}_j \geq \widetilde{d}_k$. This property ensures that queries will be processed following the FIFO policy, avoiding preemption (see Sec. 4.1). Then, we check if the query q_1 is going to miss its *modified* deadline. In such case, we set the core at maximum frequency. On the contrary, we eventually run the OYDS algorithm to select which core frequency to use. Note that we need to compute just the core frequency for the query q_1. Then, we do not need to analyze each time interval in the query queue Q. Instead, we will check only the time intervals $[t, \widetilde{d}_i] = [t, d_i - H(Q)]$ for all queries $q_i \in Q$. If a query in the queue is likely to miss its deadline, we use the maximum core

frequency to process q_1 at maximum speed. Otherwise, once we have identified the critical interval I^* (see Section 3.2) and its intensity $g(I^*)$, we select the most appropriate core frequency to process the first query q_1 by using Algorithm 2.

Algorithm 4: The PESOS algorithm for setting the most appropriate CPU core frequency to process a query

```
Data: The query queue Q and the current time t
Result: The CPU core frequency to use for processing the first query in Q
PESOS(Q, t):
1   f̄ ← max_{f∈F}{f}                                   // Maximum frequency
2   q_1 ← Q.head()                                      // First query
3   if d_1 < t then
4       return f̄
5   H(Q) ← ComputeSharedTardiness(Q, t)
6   if d_1 − H(Q) < t then
7       return f̄
8   g(I*) ← 0                                           // Maximum intensity
9   foreach query q_i in Q do
10      if d_i − H(Q) < t then
11          return f̄
12      Q_I = {q_j ∈ Q : d_j ≤ d_i − H(q)}
13      V ← Σ_{q∈Q_I} π̃_x(q)                            // Volume
14      g(I) ← V/(d_i − H(Q) − t)                       // Intensity
15      if g(I) > g(I*) then
16          g(I*) = g(I)
17  return SelectFrequency(q_1, g(I*))
```

PESOS is executed whenever a query server starts processing a new query. When the query processing is completed, the query is removed from the query queue Q. Also, PESOS is executed at each new query arrival, to take into account the increased workload in the query queue and to adjust the core frequency for the query which is currently being executed.

PESOS runs in linear time. It computes the shared tardiness using Algorithm 3, which just need to traverse the query queue. Then, the algorithm checks each interval $[t, \widetilde{d}_i]$ for all $q_i \in Q$, i.e., it analyzes $|Q|$ intervals. Eventually, it translates a processing speed into a CPU core frequency using Algorithm 2. Algorithm 2 needs to analyze at most $|F|$ CPU frequencies. In conclusion, the computational complexity of PESOS is $O(|Q| + |F|)$.

5 Experimental Setup

In this section, we firstly describe the experimental setup for the training and validation of our predictors (Sec. 5.1, Sec. 5.2). Then, we illustrate the experimental setup we adopt to measure the CPU energy consumption and the tail latency of a query processing node using our approach (Sec. 5.3). All the experiments are conducted using the Terrier search engine [29]. The platform is hosted on a dedicated server with 32 GB RAM. The operating system is Ubuntu, with Linux kernel version 3.13.0-79-generic. The machine is equipped with an Intel i7-4770K CPU, a member of the Haswell product family. The CPU has 4 physical cores which expose 15 operational frequencies $F = \{0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.1, 2.3, 2.5, 2.7, 2.9, 3.1, 3.3, 3.5\}$ GHz. The inverted index used in the experiments is obtained by indexing the ClueWeb09 (Cat. B) document collection[3] which contains more than 50 millions of Web pages. On each document, we remove stopwords and apply the Porter stemmer to all of its terms. The inverted index stores document identifiers and terms frequencies and it is kept in main memory, compressed with Elias-Fano encoding [30]. For the queries, we use the MSN 2006 query log[4].

In our experiments, we process queries using two dynamic pruning retrieval strategies: 1) MaxScore [22], and 2) WAND dynamic pruning [21]. For each query, we retrieve the top 1,000 documents according to the BM25 ranking function. The node operates with 4 query servers, i.e., processing threads, which are pinned to different CPU physical cores and share the same inverted index.

5.1 Training processing volume predictors

In this section, we adapt the *query efficiency predictors* (QEPs) introduced in [10] to originally predict the response times of a query. Instead, we modify these predictor to estimate the number of scored postings for a query. We divide queries into six query classes according to their number of terms, i.e., the first class includes queries with one term, while the last class includes queries with six or more terms.

To train and validate our predictors, we extract a number of unique queries from the MSN 2006 query log. We use unique queries to avoid any caching mechanism from the operating system that could distort our measurements. For each query class, we extract 10,000 unique queries from the MSN 2006 query log, generating a query set of 60,000 unique queries.

Before training the modified QEPs, we process each single term in the query set as detailed in [10]. We treat single terms as queries of length one. During the processing, we record the ranking scores obtained by all the documents relative to the terms, to obtain a set of 13 term-based features for each query term. Then we aggregate these to generate query-based features using three functions: maximum, variance and sum, generating a feature set containing 39 query-based aggregated features per query.

We then process the original queries in the query set to record the number of scored postings. This value is independent by the CPU frequency and we can use any $f \in F$. From the execution of the query set, we collect a processing log which contains the number of scored posting for each query in the query set. We use this processing log in the training and validation phase of the predictors.

To train our predictors, we split the feature set and the processing log: 50% of the queries for training and 50% for validation. We use the training set to learn the set of linear regressors π_x, one for each query class. Each regressors takes in input the 39 query-based aggregated features from the feature set, and estimates the number of postings scored in the processing log[5]. Note that linear regressors can return negative values for a set of input features. However, the number of scored postings is always a positive quantity. If a regressor returns a negative value, we set its prediction to the minimum between the shortest posting list length for the query terms and 1,000 (the number of retrieved document).

3. http://lemurproject.org/clueweb09/

4. http://goo.gl/ZhtnBM

5. Predictions take approximately less than 0.2 ms on average. This includes the time for computing query features, while term features are computed offline and stored in main memory.

Similarly, a linear regressor may return a value that exceeds the sum of the posting lists lengths for a query. Since this is not possible in practice, in such cases we set the prediction to the sum of the posting lists lengths.

Once we have trained the regressors on the training set, we use the validation set to see how predictors perform (results are reported in the Supplemental Material). We then use the RMSE ρ_x computed in the validation phase to correct the value of the predictors (as explained in Section 4.2). This will provide more conservative predictions to use into OYDS. The result of the training and validation phases is a set of predictors $\Pi = \{\tilde{\pi}_1, \tilde{\pi}_2, \ldots..., \tilde{\pi}_{6+}\}$.

5.2 Training processing time predictors

OYDS produces processing speeds that need to be mapped into CPU core frequencies. For this purpose, we process the 60,000 queries set described in Section 5.1 to collect the number of scored postings and the processing times of each query. From these data, we learn a set of single-variable linear regressors σ_x^f that estimate the processing time of a query given the number of its scored postings.

The processing time of a query is influenced by the CPU core frequency but also by the workload faced by the query processing node. In fact, high workloads increase the contention among the query servers (i.e., processing threads) for the main memory and the processor caches. This contention increases the time required to process a query. We want our regressors to predict processing times that match high workload conditions. This is a worst-case choice that will lead to higher energy consumption when the query processing node deals with low workloads. However, we expect to miss less query deadlines when the query processing node faces high query volumes. We process the 60,000 query set sending the to the processing node at the rate of 100 queries per second since this rate ensure than our node is constantly busy processing queries, simulating an high query workload. We process the query set 15 times, one for each frequency $f \in F$. We hence obtain 15 different processing logs reporting the number of scored postings and the processing time for each query in the query set.

Again, we divide the queries into six classes (see Sec. 5.1). For each query class and each frequency f, we learn a single-variable linear regressor σ_x^f. To learn these regressors, we split each processing log for training and validation: 50% of the logs are used for training the regressors, the remaining 50% is used to validate them. We use the validation set to check how well the predictors perform after the training phase, measuring their RMSE ρ_x^f and the coefficient of determination R^2.

Results are reported in the Supplemental Material. As expected, the mean processing times decrease by increasing the CPU frequency. Moreover the processing times are lower when using MaxScore rather than WAND. This confirms the findings in [31], [32], [33], where MaxScore outperforms WAND for memory-resident indexes.

As explained in Section 4.3, we use the RMSE R_x^f computed in the validation phase to compensate the predictors' estimates. The result of the training and validation phases is a set of predictors $\Sigma = \{\tilde{\sigma}_1^f, \tilde{\sigma}_2^f, \ldots..., \tilde{\sigma}_{6+}^f\}$.

5.3 Measuring energy consumption and tail latency

We now describe the experimental setup for measuring the CPU energy consumption and the tail latency for processing a stream of queries on a query processing node. We here focus on the tail latency since it is assumed to be a better performance indicator than the mean/median latency for Web search engines [34]. In fact, measuring the tail latency, we can affirm that most of the requests are served within the measured time interval. We require that queries are processed with a certain tail latency. We experiment with a required tail latency of 500 ms and 1,000 ms. The first value represents a scenario where we want to promptly answer the queries, while the second represents the case where we are willing to wait more time to obtain query results. In fact, search engine users are likely to not notice response delays up to 500 ms, while they are very likely to perceive delays higher than 1,000 ms [2]. In PESOS we can impose the tail latency constrain setting $\tau = \{500, 1,000\}$ ms, i.e., requiring that queries are processed within τ ms since their arrival. We test different latency requirements to observe if PESOS can produce energy savings while meeting the required tail latency. The query processing is performed using the MaxScore and the WAND retrieval strategies, to understand how PESOS behaves when different retrieval strategies are deployed. Also, we test PESOS with predictors corrected using their RMSE (as discussed in Sec. 4.2 and 4.3), and without any correction. We will refer to the first configuration as *time conservative* (TC) and to the second as *energy conservative* (EC). In the TC configuration, we are likely to over-estimate the processing volume and time of some queries, requiring higher core frequencies. However, we also expect to miss less query deadlines hence producing lower tail latencies. In the EC configuration, instead, we use predictors without any correction which should lead to lower core frequencies and produce higher energy savings. Comparing the two configurations, we want to understand if acceptable tail latencies are achievable even without predictors correction.

To perform our measurements, we carry out two different kinds of experiment. Firstly, we observe the behavior of PESOS under a synthetic query workload. For this purpose, we send a stream of 60,000 unique queries from the MSN2006 log to the processing node. Table 1 shows the number of queries for each query class, with an average of ∼3 terms per query. This value reflects the average query length observable on the original MSN2006 log. To test the robustness of PESOS, we experiment with different query arrival rates, i.e., {5, 10, 15, 20, 25, 30, 35} query per second (QPS) sent to the processing node[6]. The second kind of experiment aims to observe the behavior of PESOS under a realistic query workload. For this, we process 544,718 unique queries from the MSN2006 query log following the actual query arrivals of the second day of the query log. Table 1 reports the number of queries for each query class, while Figure 3 show the number of query arrivals during the day. For both query workloads, we process unique queries to avoid caching mechanism that could compromise the evaluation of the experiment results. Nevertheless, for the realistic query workload we are still processing the same

6. Note that the τ and QPS values can be rescaled by considering smaller inverted indexes, for instance when the index is partitioned across multiple query processing nodes.

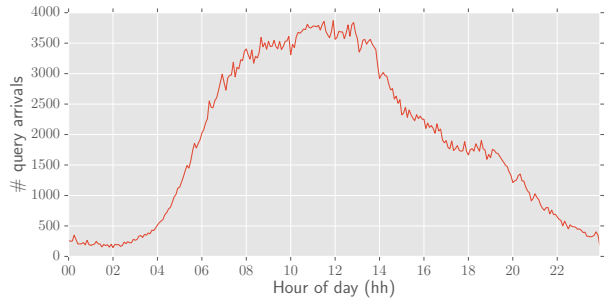

Fig. 3. Query arrivals for the second day of the MSN2006 query log, aggregated every 5 minutes.

TABLE 1
Distribution of queries across the various query classes for the synthetic and the realistic query sets

	1	2	3	4	5	6+
Synthetic	5,644	17,871	18,913	10,828	4,331	2,413
Realistic	51,553	161,973	171,016	98,001	39,998	22,177

number of queries reported in the second day of the MSN2006 query log to reflect a realistic query traffic.

Finally, we compare the energy consumption and the tail latency of PESOS against three baselines, namely `perf`, `power`, and `cons`. `perf` and `power` are provided by the `intel_pstate` driver [8]. The `perf` policy simply uses the highest core frequency to process queries and then race to an idle state. The `power` policy, instead, selects the frequency for a core according to its utilization. High frequencies are selected when a core is highly utilized. Conversely, lower frequencies are selected when a core is lowly utilized. Differently, the `cons` policy [13] bases its decisions upon the utilization of a query server rather than on the utilization of a CPU core. The utilization of a query server is computed as the ratio between the query arrival rate and service rate. The frequency of a core is then throttled if the server utilization is above 80% or below 20%, to produce a desirable utilization of 70%. The `cons` policy executes every 2 seconds. We select these parameter settings to achieve the best energy savings while maintaining acceptable latencies, reflecting those used in [13].

With these experiments we want to address the following research questions:

- RQ1: Does PESOS meet the required tail latencies?
- RQ2: Does PESOS help reducing the CPU energy consumption of a query processing node?
- RQ3: Is prediction correction necessary to achieve acceptable tail latencies?
- RQ4: How does PESOS behave using different retrieval strategies, with different prediction accuracies?

We measure the 95-th percentile tail latency of the processing node to answer our first research question. The 95-th percentile tail latency is used to measure the effects of power management mechanism on the responsiveness of search systems in [15], [16]. To answer the second research question we measure the energy consumption of the CPU using the Mammut library[7] which relies on the Intel Running Average Power Limit (RAPL) interface. The RAPL component performs actual measurements of the energy consumption in Haswell processors. Hackenberg et al. [35] show the reliability of such measurements, and the RAPL interface is used in other works to measure the energy consumption of CPUs [36], [37]. Finally, to address the third research question we compare the performance of our approach with and without prediction corrections. We compare the performance of PESOS with MaxScore and WAND to answer the last research question. All experiments are conducted using the query processing node described at the beginning of this Section.

7. http://danieledesensi.github.io/mammut/

6 Results

In this Section we discuss the results of our experiments. We firstly describe the results relatively to the experiments conducted with synthetic query workloads. Then, we illustrate the results obtained using the realistic query workload.

6.1 Synthetic query workload results

We begin by analyzing the behavior of `perf` and `power`. We recall that `perf` always uses the maximum available CPU core frequency, while `power` is an utilization-based policy which throttles a CPU core frequency accordingly to its utilization. Both `perf` and `power`, however, do not permit to impose the required tail latency of a query processing node. From Table 2 we can observe that, when MaxScore is deployed, `perf` meets the 500 ms tail latency requirement up to 30 QPS, while the 1,000 ms tail latency requirement is always satisfied. When WAND is used, instead, `perf` satisfies the 500 ms tail latency up to 20 QPS, and the 1,000 ms tail latency up to 30 QPS. We explain this difference by recalling that WAND provides longer response times than MaxScore (see Table 2 in Supplemental Material). With respect to tail latencies, we observe a similar behavior between `perf` and `power`. This is expected since, as the query arrival rate increases, the CPU cores utilization increases as well, leading `power` to select high core frequencies and hence behaving like `perf`. In terms of energy savings[8], Table 3 shows little differences between the two baselines. Some energy savings are provided by `power` at low QPS, from ∼2% in the case of WAND up to ∼5% for MaxScore, at the cost of higher tail latency. For high query arrival rates, `power` can be even detrimental, increasing the energy consumption of the system. We explain this behavior with the longer query processing times and the overhead introduced by the policy, i.e., the CPU cores spend more time busy doing computations, hence consuming more energy.

Regarding the other baseline, we observe in Table 2 that `cons` satisfies the 500 ms tail latency only for moderate QPS (from 15 to 25) when MaxScore is deployed, and only for 20-25 QPS with WAND. Again, this is due to the better performance of MaxScore over WAND. When considering a tail latency of 1000 ms, we observe that `cons` meets the latency requirement from 10 to 35 QPS with MaxScore and from 10 to 30 QPS with WAND. In general, we can conclude that `cons` produces latency violations when the query arrival rate is particularly low or high. We explain this behavior by recalling that `cons` requires to tune several parameters which

8. Energy consumption decreases as the query arrival rate increases, since experiments take less time to complete.

drive its decisions about frequency scaling. In our experiments we use a setting aimed to produce the best energy savings and acceptable latencies. However, our results suggests that a single parameter setting is not sufficient for `cons` to perform well under a wide range of query arrival rates. With respect to energy consumption, Table 3 shows that `cons` provides substantial energy savings with respect to `perf` at low QPS (up $\sim 45\%$ with Maxscore and $\sim 40\%$ with WAND). However, when the query arrival rate increases, `cons` can consume more energy. Again, we explain this behavior with the longer query processing times and the overhead introduced by the policy.

We now discuss the results for PESOS when using $\tau = 500$ ms and $\tau = 1,000$ ms. For the time conservative configuration, Table 2 shows that PESOS satisfies the 500 ms tail latency requirement from 5 to 20 QPS when using WAND and up to 25 QPS when using MaxScore. For the 1,000 ms tail latency requirement, in the time conservative configuration PESOS meets the required latency up to 30 QPS for both retrieval strategies. These results are similar to what reported for the `perf` policy. Relatively to our first research question (RQ1), we can state that PESOS is able to meet the required tail latencies for the same query workloads sustainable by a system which operates at maximum CPU core frequency.

In terms of energy savings, Table 3 shows that PESOS markedly reduce the energy consumption of the query processing node's CPUs. In the time conservative configuration, PESOS can reduce the energy consumption up to $\sim$25% when using MaxScore and up to $\sim$12% when using WAND. We explain the better results achieved with MaxScore with the higher accuracy of its processing time predictors compared to the ones for WAND (see Table 2 in Supplemental Material). We also notice that energy savings diminish as the query arrival rate increases, as there are less opportunities for PESOS to use low core frequencies without violating query deadlines. Relatively to our second research question (RQ2), the results in Table 3 show that PESOS actually permits to reduce the CPU energy consumption of a query processing node. In most cases, these energy savings are higher than those provided by the state-of-the-art `power` and `cons` policies. This indicates that application-dependent information leveraged by PESOS, such as the state of the query queues and the query efficiency predictors, are a better input for managing the CPU cores frequencies than the cores or query servers utilizations. Also, an important role is played by the τ parameter, which permits to set the required tail latencies rather than processing the queries at maximum speed as in `perf`, which does not take into account latency requirements.

We now analyze the performance of PESOS in the energy conservative configuration, i.e., when we do not correct the query efficiency predictors using their RMSE. Table 2 shows that, for both retrieval strategies, PESOS misses the 500 ms tail latency requirement. This answer our third research question (RQ3): predictors correction is necessary to meet the latency requirements. However, we highlight that the reported latency violations are limited: for the same QPS values for which the time conservative configuration meets the 500 ms tail latency requirement, the energy conservative configurations violates the requirement by up to $\sim$8% with WAND and up to $\sim$15% with MaxScore. Additionally, we notice higher energy savings compared to the time conservative configuration (see Table 3). When $\tau = 500$ ms, the energy conservative configuration reduces the energy consumption of the CPU node by $\sim$29% in the case of WAND and by $\sim$34% in the case of MaxScore for low QPS. In Table 2 we can observe that the 1,000 ms tail latency requirement is met up to 30 QPS when MaxScore is applied, and up to 25 QPS when WAND is used. This suggests that predictors correction becomes less relevant as the latency requirement increases. Remarkably, the energy conservative configuration basically halves the energy consumption of the CPU node for 5 QPS when $\tau = 1,000$ ms (see Table 3).

Finally, to answer our last research question (RQ4), we compare the behavior of PESOS while deploying MaxScore and WAND. In general, PESOS shows better results with MaxScore. In fact, the tail latency requirements are met for slightly higher QPS values compared to WAND. Also, PESOS shows higher energy savings when the MaxScore retrieval strategy is applied. We explain this behavior with the faster response time provided by MaxScore and by the higher precision of its processing time predictors.

6.2 Realistic query workload results

Now we describe the results of the experiments conducted processing the realistic query workload. In this subsection we will not investigate research question RQ4 as for these experiments we use only the MaxScore retrieval strategy, which provided the best results in Section 6.1. Firstly, we will analyze the performance of the three baselines. Then, we will discuss the results obtained by PESOS in the time conservative configuration. Finally, we will study the performance of PESOS in the energy conservative configuration.

Figure 4 reports the tail latencies of the tested approaches during the day. As expected, `perf` provides lower latencies than the other approaches. Unsurprisingly, `perf` exhibits also the higher CPU energy consumption as it always uses the maximum core frequency (see Tab. 4). In terms of tail latency, `power` behaves similarly to `perf` during midday but exhibits higher latencies at the beginning and at the end of the day. This behavior is explained in Figure 5 (left). During midday, the CPU cores are highly utilized due to the higher number of query arrivals. In response to high core utilization, `power` selects the maximum core frequency as in `perf`. During the rest of the day, instead, the query arrivals decrease and the CPU cores are less utilized. Therefore, `power` selects lower core frequencies which explain longer latencies. For the same reasons, `power` provides limited energy savings compared to `perf`, reducing the CPU energy consumption by less than 4% as reported in Table 4. Figure 6 illustrate the energy reductions of `power` with respect to `perf` during the day. When `power` is applied, we can observe energy savings only at the beginning and at the end of the day, when `power` selects lower core frequencies as shown in Figure 5 (left). In these periods, the CPU consumes $\sim$20% less energy with respect to `perf`. However, during midday `power` does not provide any energy saving. Again, this is due to the high utilizations showed by the CPU cores during midday In this situation, `power` selects the maximum core frequency, behaving like `perf` and consuming the same amount of energy.

Table 4 shows that `cons` can reduce by $\sim$27% the CPU energy consumption with respect to `perf`. As shown in Figure 6, energy consumption can be reduced by $\sim$45% during periods

TABLE 2
MaxScore (left) and WAND (right) tail latencies (95th-tile, in ms) of baselines, time conservative (TC), and energy conservative (EC) PESOS for different synthetic query workload (QPS)

QPS	Baselines			PESOS $\tau = 500$ ms		PESOS $\tau = 1,000$ ms	
	perf	power	cons	TC	EC	TC	EC
MaxScore							
5	342	360	1,019	446	573	809	980
10	344	344	667	431	536	759	894
15	341	346	442	428	509	703	833
20	362	364	393	415	489	685	832
25	402	400	411	446	500	701	842
30	479	498	515	522	563	835	948
35	657	715	687	725	731	1,174	1,287

QPS	Baselines			PESOS $\tau = 500$ ms		PESOS $\tau = 1,000$ ms	
	perf	power	cons	TC	EC	TC	EC
WAND							
5	378	399	1,060	399	538	649	896
10	380	382	714	389	510	615	813
15	391	396	519	401	490	586	757
20	437	436	457	439	502	585	765
25	527	537	546	534	569	627	793
30	821	835	787	821	867	884	1,035
35	2,696	3,091	2,831	3,211	3,585	2,667	3,318

TABLE 3
Energy consumption (KJ) of baselines, time conservative (TC), and energy conservative (EC) PESOS, with energy savings w.r.t. perf for different synthetic query workload (QPS)

QPS	Baselines					PESOS $\tau = 500$ ms				PESOS $\tau = 1,000$ ms			
	perf	power		cons		TC		EC		TC		EC	
MaxScore													
5	92.79	87.78	(-5.40%)	51.06	(-44.97%)	69.95	(-24.62%)	61.34	(-33.89%)	47.43	(-48.89%)	42.56	(-54.13%)
10	83.51	81.35	(-2.58%)	58.32	(-30.16%)	65.38	(-21.71%)	57.30	(-31.39%)	47.36	(-43.29%)	44.81	(-46.34%)
15	77.78	77.54	(-0.31%)	74.33	(-4.44%)	64.35	(-17.26%)	57.26	(-26.37%)	50.22	(-35.43%)	48.74	(-37.34%)
20	75.37	75.34	(-0.05%)	75.01	(-0.48%)	62.21	(-17.46%)	59.42	(-21.17%)	52.35	(-30.55%)	52.65	(-30.15%)
25	72.75	73.09	(0.47%)	74.23	(2.03%)	65.77	(-9.59%)	62.57	(-13.99%)	56.46	(-22.39%)	56.74	(-22.01%)
30	70.74	71.43	(0.98%)	72.61	(2.64%)	66.50	(-6.00%)	65.06	(-8.04%)	62.42	(-11.76%)	65.01	(-8.10%)
35	69.78	71.51	(2.48%)	71.53	(2.51%)	68.46	(-1.89%)	66.89	(-4.14%)	70.02	(0.35%)	68.70	(-1.55%)
WAND													
5	106.49	104.28	(-2.07%)	64.11	(-39.80%)	93.83	(-11.89%)	76.03	(-28.60%)	67.38	(-36.72%)	56.48	(-46.96%)
10	96.62	95.25	(-1.42%)	74.13	(-23.28%)	88.01	(-8.91%)	74.66	(-22.73%)	67.11	(-30.54%)	60.19	(-37.70%)
15	91.55	91.98	(0.46%)	87.27	(-4.66%)	84.56	(-7.64%)	75.80	(-17.21%)	68.39	(-25.30%)	64.27	(-29.81%)
20	89.34	89.31	(-0.04%)	89.27	(-0.08%)	83.49	(-6.55%)	78.44	(-12.20%)	72.11	(-19.29%)	70.72	(-20.84%)
25	85.81	86.59	(0.91%)	87.17	(1.58%)	83.69	(-2.47%)	79.32	(-7.56%)	75.96	(-11.48%)	73.92	(-13.86%)
30	85.27	86.38	(1.31%)	85.85	(0.68%)	84.37	(-1.05%)	82.03	(-3.80%)	80.82	(-5.22%)	80.03	(-6.14%)
35	84.58	84.86	(0.34%)	85.72	(1.35%)	84.70	(0.15%)	84.88	(0.36%)	82.72	(-2.20%)	84.15	(-0.50%)

of low query workloads. However, such energy savings come at the price of high latencies (see Fig. 4). Indeed, cons exhibits tail latencies that are above 500 ms during midday, and above 1,000 ms during the rest of the day. In fact, cons relies on low core frequencies most of the time (see Fig. 5 (middle)), hence producing long query response times. It starts selecting higher core frequency only during midday, when the query servers utilizations increases due to the higher query workload. In fact, we can notice in Figure 4 how tail latencies reduces during midday. The results reported in this section confirm those presented in Section 6.1, where cons poorly behaves in presence of low query arrival rates. Indeed, cons requires a careful parameter tuning, and a static parameter setting cannot efficiently cope with query arrivals rate that widely varies throughout the day.

Regarding time conservative PESOS, we can observe in Figure 4 that the 500 ms tail latency requirement is successfully met, with very few violations. Similarly, the time conservative configuration is able to meet the 1,000 ms tail latency requirement, remaining well below the required threshold. Relatively to our first research question (RQ1), we conclude that PESOS can successfully meet the required tail latency when the time conservative configuration is applied, even for a realistic query workload. At the same time, PESOS shows significant energy savings with respect to perf, as reported in Table 4. In fact, with a 500 ms tail latency requirement, time conservative PESOS reduces the CPU energy consumption by ∼24%, and by ∼44% when we impose a 1,000 ms tail latency

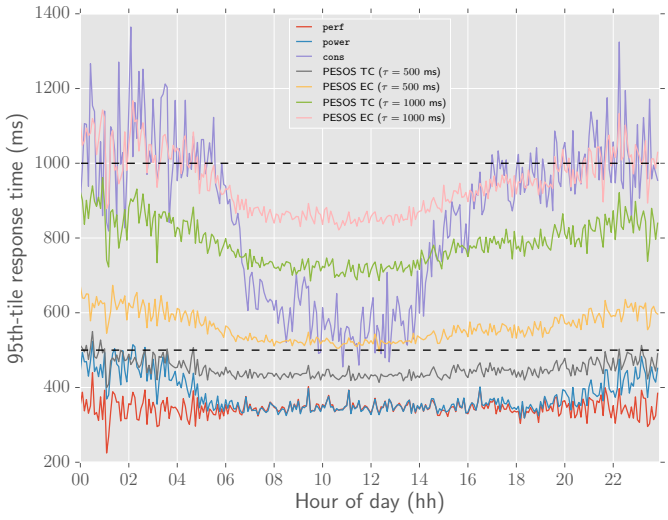

Fig. 4. Tail latencies during a day, aggregated every 5 minutes.

requirement. As shown in Figure 6, such energy savings are present during the whole day, up to ∼30% under the 500 ms tail latency requirement, and ∼50% for the 1,000 ms tail latency requirement. These energy savings are possible thanks to the application-level information exploited by the PESOS algorithm, such as the states of the query queues and the query efficiency predictions. Also, an important role is played

by the τ parameter, which permits to set the required tail latency. As we can see from Figure 5 (right), this information permits PESOS to select lower core frequencies more often than `power`, which takes its frequency scaling decision relying only on the CPU cores utilizations. High core frequencies are selected by time conservative PESOS only in limited cases during midday, when the query load is more intense. Relatively to our second research question (RQ2), we can then conclude that PESOS successfully reduces the CPU energy consumption of a query processing nodes, providing much higher energy savings than the `power`. At the same time, PESOS provides energy savings comparable to those produce by `cons`, while incurring in many less latency violations.

We now analyze the results for PESOS in its energy conservative configuration. As shown in Figure 4, energy conservative PESOS does not satisfy the tail latency requirements. However, we notice that the tail latency of energy conservative PESOS approaches the 500 ms requirement during midday, when the query load is more intense and the query queues are populated with an higher number of queries than in other periods of the day. In the PESOS algorithm, this results in critical intervals of high intensity (see Alg. 4) which lead PESOS to select higher core frequencies, hence reducing the tail latency of the system. We can observe the same effects when we impose a 1,000 ms tail latency requirement. In this case, the energy conservative configuration violates the requirement at the beginning and at the end of the day, when the query workload is not intense. On the contrary, the 1,000 ms tail latency requirement is met during midday in correspondence of an high query arrival rate. While violating the tail latency requirements, the energy conservative configurations provides the highest energy savings as reported in Table 4. When we impose a tail latency requirement of 500 ms, energy conservative PESOS reduces the CPU energy consumption by almost 33% compared to `perf`. The energy savings reach ∼48% when the tail latency requirement is set to 1,000 ms. Such savings are present during the whole day, as illustrated in Figure 6, up to ∼40% under the 500 ms tail latency requirement, and ∼60% for the 1,000 ms tail latency requirement. Interestingly, we notice a larger energy saving gap between time conservative and energy conservative PESOS when $\tau = 500$ ms than when $\tau = 1,000$ ms. This is particular evident in Figure 6, where the curves relative to the two configurations almost coincides. This is surprising, as time conservative PESOS is likely to over estimate the processing volumes and times for some queries, selecting higher core frequencies and consuming more energy. However, this behavior can be due to the longer time budgets available to process queries under the 1,000 ms latency constraint, which still permits time conservative PESOS to select lower core frequencies to process queries. Regarding our third research question (RQ3), we can then conclude that predictors correction is necessary to meet the required tail latencies and overall time conservative PESOS is a better choice when processing a realistic query workload.

TABLE 4
CPU energy consumption (KJ) of the power management approaches for processing a day of query log, and the gain w.r.t. `perf`

	Energy (KJ)	Gain (%)
`perf`	790.40	–
`power`	759.42	-3.92%
`cons`	575.49	-27.19%
PESOS (TC, $\tau = 500$ ms)	601.67	-23.88%
PESOS (EC, $\tau = 500$ ms)	531.10	-32.81%
PESOS (TC, $\tau = 1,000$ ms)	443.73	-43.86%
PESOS (EC, $\tau = 1,000$ ms)	412.06	-47.87%

7 Related Work

While Web search engines can consume tens of megawatts of electric power to operate [1], there is only a limited body of research that aims to reduce the energy expenditure of Web search engines. These works can be divided in three categories which focus on different level of a Web search engine architecture: 1) geographically distributed datacenters, 2) processing clusters within a datacenter, and 3) a single query processing node.

The works in [38], [39], [40] focus on multi-site Web search engines, i.e., search engines composed by multiple and geographically distant datacenters. These studies propose to use *query forwarding*, i.e., to shift the query workload between datacenters. Kayaaslan et al. [38] consider a scenario where datacenters hold the same replica of the inverted index. They propose to use query forwarding to exploit the difference in energy price at different sites, due to the different datacenter locations and timezones. In this way, they aim to minimize the energy expenditure of the search engine. At the same time, the approach ensures that the remote sites can process forwarded queries without exceeding their processing capacity. Blanco et al. [39] extend this idea by forwarding queries towards datacenters that can use *renewable energy sources* that are both environmentally friendly and economically convenient. Teymorian et al. [40], instead, consider a scenario where each site hold a different inverted index. In their approach, the authors use query forwarding to maximize the quality of search results, collecting relevant document from the different sites, while satisfying energy cost budget constraints. Query forwarding techniques may be applied in conjunction with PESOS to deploy more energy-efficient architectures.

The works in [15], [41], [42] focus on reducing the energy consumption of query processing node clusters within a single datacenter. Sazoglu et al. [41] investigate the role of result caching in the energy expenditure of search engines. They present a financial cost metric to measure the price of cache misses and find that cost-aware caching strategies can reduce the energy expenditure of a datacenter when there is an high variation of energy prices during the day. Freire et al. [42] propose a self-adaptive model that exploits the historical and current query loads of the system. The model autonomously decide whether to activate a query processing node, to provide acceptable query response times, or put it in standby to save energy. Lo et al. [15], instead, introduce a feedback-based model that dynamically cap the power consumption of query processing nodes CPUs. Their approach trades off power savings for longer latencies that barely meet the response time requirements under any query workload. We believe that PESOS can be used together with the techniques described in [41], [42] to improve the energy efficiency of a Web search engine. On the contrary, the integration of PESOS with the approach proposed in [15] needs to be investigated, since both techniques require to control the CPU power management.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2017.2681279, IEEE Transactions on Knowledge and Data Engineering

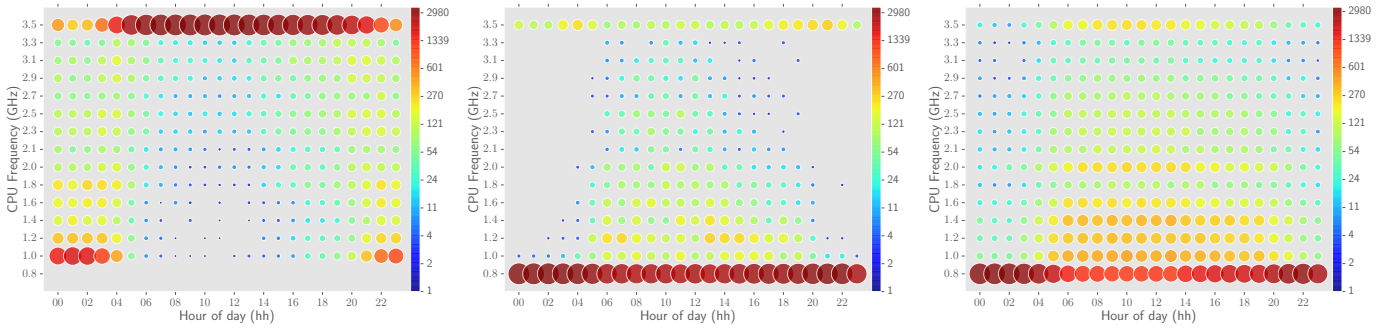

Fig. 5. Number of times power (left), cons (middle) and time-conservative ($\tau = 500$ ms) PESOS (right) select frequencies on one of the CPU cores during the day, sampled every second.

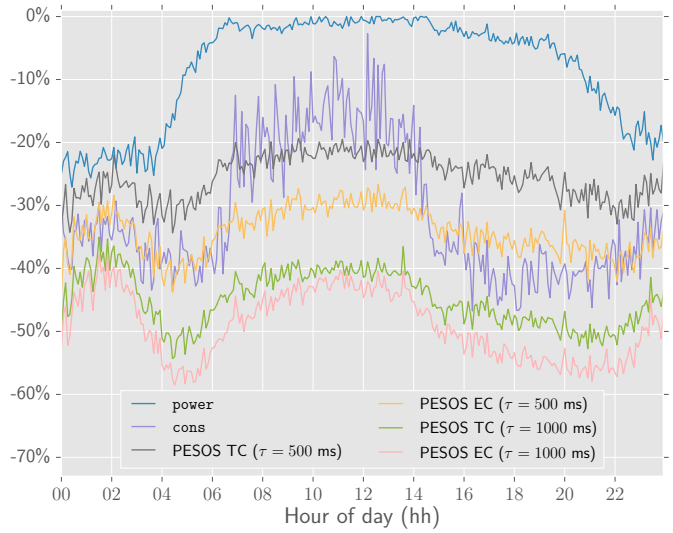

Fig. 6. CPU energy reductions of power and PESOS w.r.t perf, aggregated every 5 minutes.

Finally, the works in [13], [43] focus on reducing the energy consumption of a single query node. Catena et al. [13] propose to use the query processing node utilization, rather than the CPU utilization, to accordingly throttle the CPU frequency and reduce the power consumption of the node. Du et. al [43], propose an approach to improve the energy efficiency of a query node by equally distribute queries and power among the CPU cores. However, their work contemplates the early termination of query processing, possibly degrading the quality of the search results. In our work, instead, queries are always completely processed, even if this may delay the execution of other queries. Also, the approaches in [13], [43] do not consider the characteristics of the incoming queries, i.e., differently from PESOS, no form of query efficiency prediction is applied to achieve energy savings.

8 Conclusions

In this paper we proposed the Predictive Energy Saving Online Scheduling (PESOS) algorithm. In the context of Web search engines, PESOS aims to reduce the CPU energy consumption of a query processing node while imposing a required tail latency on the query response times. For each query, PESOS selects the lowest possible CPU core frequency such that the energy consumption is reduced and the deadlines are respected. PESOS selects the right CPU core frequency exploiting two different kinds of query efficiency predictors (QEPs). The first QEP estimates the processing volume of queries. The second QEP estimates the query processing times under different core frequencies, given the number of postings to score. Since QEPs can be inaccurate, during their training we recorded the root mean square error (RMSE) of the predictions. In this work, we proposed to sum the RMSE to the actual predictions to compensate prediction errors. We then defined two possible configuration for PESOS: *time conservative*, where prediction correction is enforced, and *energy conservative*, where QEPs are left unmodified.

We experimentally evaluated the performance of PESOS using the ClueWeb09B corpus and processing queries from the MSN2006 log applying two different dynamic pruning retrieval strategies: MaxScore and WAND. We compared the performance of PESOS with those of three baselines: `perf`, which always uses the maximum CPU core frequency, `power`, which throttles frequencies according to the core utilizations, and `cons`, which throttles frequencies according to the utilization of the query servers. We found that time conservative PESOS was able to meet a required tail latency of 500 and 1,000 ms for the same workload sustainable by `perf`. At the same time, time conservative PESOS was able to reduce the CPU energy consumption of the CPU by ∼12% with WAND up to ∼25% with MaxScore, for which we could train more accurate query efficiency predictors than for WAND. Greater energy savings were observable with energy conservative PESOS, but at the cost of higher latencies. Predictors correction is hence necessary to obtain the required tail latency, still providing significant energy savings. Moreover, we processed a realistic query workload which reflects the query arrivals of one day of the MSN2006 log. We found that time conservative PESOS was able to meet a 500 ms (with very few violations) and a 1,000 ms tail latency requirements, while reducing the CPU energy consumption, respectively, by ∼24% and by ∼44% when compared to `perf`. From the same set of experiments, we reported that `power` can reduce the CPU energy consumption by just ∼4% with respect to `perf`. On the other hand, `cons` was able to reduce the CPU energy consumption by ∼27% but incurring in considerable latency violations. We justified the superior `perf` provided by PESOS thanks to the application-level information exploited by our algorithm, such as the knowledge about the state of the query queues and the query efficiency predictions.

References

[1] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2nd ed. Morgan & Claypool Publishers, 2013.

[2] I. Arapakis, X. Bai, and B. B. Cambazoglu, "Impact of response latency on user behavior in web search," in *Proc. SIGIR*, 2014, pp. 103–112.

[3] U.S. Department of Energy, "Quick start guide to increase data center energy efficiency," 2009. [Online]. Available: http://goo.gl/ovDP26

[4] The Climate Group for the Global e-Sustainability Initiative, "Smart 2020: Enabling the low carbon economy in the information age," 2008. [Online]. Available: http://goo.gl/w5gMXa

[5] European Commission - Joint Research Centre, "The European Code of Conduct for Energy Efficiency in Data Centre." [Online]. Available: http://goo.gl/wmqYLQ

[6] U.S. Department of Energy, "Best Practices Guide for Energy-Efficient Data Center Design." [Online]. Available: http://goo.gl/pikFFv

[7] D. C. Snowdon, S. Ruocco, and G. Heiser, "Power Management and Dynamic Voltage Scaling: Myths and Facts," in *Proc. of Workshop on Power Aware Real-time Computing*, 2005.

[8] The Linux Kernel Archives, "Intel P-State driver." [Online]. Available: https://goo.gl/w9JyBa

[9] D. Brodowski, "CPU frequency and voltage scaling code in the Linux kernel." [Online]. Available: https://goo.gl/QSkft2

[10] C. Macdonald, N. Tonellotto, and I. Ounis, "Learning to predict response times for online query scheduling," in *Proc. SIGIR*, 2012, pp. 621–630.

[11] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Predictive parallelization: Taming tail latencies in web search," in *Proc. SIGIR*, 2014, pp. 253–262.

[12] S. Kim, Y. He, S.-w. Hwang, S. Elnikety, and S. Choi, "Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search," in *Proc. WSDM*, 2015, pp. 7–16.

[13] M. Catena, C. Macdonald, and N. Tonellotto, "Load-sensitive cpu power management for web search engines," in *Proc. SIGIR*, 2015, pp. 751–754.

[14] V. Pallipadi, S. Li, and A. Belay, "cpuidle: Do nothing, efficiently," in *Proc. Linux Symposium*, vol. 2, 2007, pp. 119–125.

[15] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proc. ISCA*, 2014, pp. 301–312.

[16] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proc. ISCA*, 2011, pp. 319–330.

[17] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[18] M. Catena, C. Macdonald, and I. Ounis, "On inverted index compression for search engine efficiency," in *Proc. ECIR*, 2014, pp. 359–371.

[19] J. Dean, "Challenges in building large-scale information retrieval systems: Invited talk," in *Proc. WSDM*, 2009.

[20] S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," *Found. Trends Inf. Retr.*, vol. 3, no. 4, pp. 333–389, Apr. 2009.

[21] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien, "Efficient query evaluation using a two-level retrieval process," in *Proc. CIKM*, 2003, pp. 426–434.

[22] H. Turtle and J. Flood, "Query evaluation: Strategies and optimizations," *Inf. Process. Manage.*, vol. 31, no. 6, pp. 831–850, Nov. 1995.

[23] H. Wu and H. Fang, "Analytical performance modeling for top-k query processing," in *Proc. CIKM*, 2014, pp. 1619–1628.

[24] A. Freire, C. Macdonald, N. Tonellotto, I. Ounis, and F. Cacheda, "Hybrid query scheduling for a replicated search engine," in *Proc. ECIR*, 2013, pp. 435–446.

[25] S. Albers, F. Müller, and S. Schmelzer, "Speed scaling on parallel processors," in *Proc. SPAA*, 2007, pp. 289–298.

[26] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *Proc. FOCS*, 1995, pp. 374–382.

[27] N. Bansal, T. Kimbrel, and K. Pruhs, "Speed scaling to manage energy and temperature," *J. ACM*, vol. 54, no. 1, pp. 3:1–3:39, Mar. 2007.

[28] S. Albers, "Online scheduling," *Introduction to Scheduling*, pp. 57–84, 2009.

[29] C. Macdonald, R. McCreadie, R. L. Santos, and I. Ounis, "From puppy to maturity: Experiences in developing terrier," *Open Source Information Retrieval*, vol. 60, 2012.

[30] S. Vigna, "Quasi-succinct indices," in *Proc. WSDM*, 2013, pp. 83–92.

[31] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Y. Zien, "Evaluation strategies for top-k queries over memory-resident inverted indexes," *PVLDB*, vol. 4, no. 12, pp. 1213–1224, 2011.

[32] G. Ottaviano, N. Tonellotto, and R. Venturini, "Optimal space-time tradeoffs for inverted indexes," in *Proc. WSDM*, 2015, pp. 47–56.

[33] C. Dimopoulos, S. Nepomnyachiy, and T. Suel, "Optimizing top-k document retrieval strategies for block-max indexes," in *Proc. WSDM*, Rome, Italy, 2013, pp. 113–122.

[34] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[35] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the intel haswell processor," in *Proc. IPDPSW*, 2015, pp. 896–904.

[36] D. De Sensi, "Predicting performance and power consumption of parallel applications," in *Proc. PDP*, 2016, pp. 200–207.

[37] M. Danelutto, D. De Sensi, and M. Torquati, "Energy driven adaptivity in stream parallel computations," in *Proc. PDP*, 2015, pp. 103–110.

[38] E. Kayaaslan, B. B. Cambazoglu, R. Blanco, F. P. Junqueira, and C. Aykanat, "Energy-price-driven query processing in multi-center web search engines," in *Proc. SIGIR*, 2011, pp. 983–992.

[39] R. Blanco, M. Catena, and N. Tonellotto, "Exploiting green energy to reduce the operational costs of multi-center web search engines," in *Proc. WWW*, 2016, pp. 1237–1247.

[40] A. Teymorian, O. Frieder, and M. A. Maloof, "Rank-energy selective query forwarding for distributed search systems," in *Proc. CIKM*, 2013, pp. 389–398.

[41] F. B. Sazoglu, B. B. Cambazoglu, R. Ozcan, I. S. Altingovde, and O. Ulusoy, "A financial cost metric for result caching," in *Proc. SIGIR*, 2013, pp. 873–876.

[42] A. Freire, C. Macdonald, N. Tonellotto, I. Ounis, and F. Cacheda, "A self-adapting latency/power tradeoff model for replicated search engines," in *Proc. WSDM*, 2014, pp. 13–22.

[43] Z. Du, H. Sun, Y. He, Y. He, D. A. Bader, and H. Zhang, "Energy-efficient scheduling for best-effort interactive services to achieve high response quality," in *Proc. IPDPS*, 2013, pp. 637–648.

Matteo Catena received the BS and MS degrees in Computer Science from the University of L'Aquila in 2010 and 2013, respectively. He is now a PhD student at the Gran Sasso Science Institute and a research associate at ISTI-CNR. His main research interests include Web information retrieval, Green computing and compression algorithms.

Nicola Tonellotto received the PhD degrees in Computer Engineering from the University of Pisa and the Technical University of Dortmund in 2008. He is a researcher at National Research Council of Italy. His main research interests include cloud computing, resource management and Web information retrieval. He has authored more than 50 papers on these topics in peer reviewed international journal and conferences. He is a member of the ACM and SIGIR.