Energy and Runtime Performance Optimization of Node.js Web Requests

Conference Paper · October 2021

DOI: 10.1109/IC2E52221.2021.00021

4 authors, including:

Maria Patrou
University of New Brunswick
9 PUBLICATIONS **14** CITATIONS
SEE PROFILE

Kenneth B. Kent
University of New Brunswick
140 PUBLICATIONS **1,052** CITATIONS
SEE PROFILE

Michael Dawson
IBM
27 PUBLICATIONS **105** CITATIONS
SEE PROFILE

Energy and Runtime Performance Optimization of Node.js Web Requests

Maria Patrou and Kenneth B. Kent
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
maria.patrou@unb.ca, ken@unb.ca

Joran Siu
IBM Runtime Technologies
IBM
Toronto, Canada
joransiu@ca.ibm.com

Michael Dawson
Node.js Lead for Red Hat and IBM
Red Hat
Ottawa, Canada
midawson@redhat.com

***Abstract*—The Node.js framework uses an event-driven model with a single-threaded event loop and provides asynchronous and non-blocking I/O operations. As with other programs, Node.js web applications take advantage of underlying resources, including CPUs, which can incorporate the dynamic voltage and frequency scaling (DVFS) technique. Using CPU DVFS, the applications can increase their runtime performance, at the expense of the system's energy consumption. Thus, software code that utilizes the CPU DVFS technique efficiently should lead to "green" and high-performing applications with respect to the business logic. To this end, we build a CPU frequency scaling/energy aware system to enable CPU frequency control within Node.js applications and measure the energy consumption of specific tasks. We also build a benchmark suite to analyze the energy consumption and runtime performance of different requests based on the CPU frequency impact and collect information and patterns, as we scale the CPU frequency. The analysis aims to provide data and knowledge on the CPU frequency "suitability" and impact in order to create a model for CPU frequency scaling on Node.js web applications and achieve an efficient and sustainable runtime performance.**

***Index Terms*—Node.js, CPU DVFS, green energy, efficiency**

I. Introduction

Node.js applications, such as web applications and web servers, can scale up and parallelize their operations to use all the available resources and improve their runtime performance. However, as more resources are utilized, such as CPUs, machines consume more energy to meet the increased demand. For example, an approximate peak power usage distribution of warehouse-scale machines (WSCs) deployed at Google in 2017 reveals that CPUs consume 61% of the total energy, followed by DRAM with 12% [1]. Although, the numbers can vary on different occasions, CPUs can be an energy dominant component and, thus they are an important factor to investigate for energy management.

Modern processors use the dynamic voltage and frequency scaling (DVFS) technique to reduce the frequency and voltage of CPUs and achieve power reduction. The DVFS technique introduces several operating performance points, different frequency and voltage pairs, affecting the system's performance and the energy consumption. Systems use the DVFS technique in various levels, including OS and application, to scale the CPU frequency/voltage and achieve better efficiency. They use metrics and models to measure, analyze, predict and apply suitable CPU frequencies based on the current workload.

However, we can incorporate the CPU DVFS technique within Node.js web applications to enable dynamic control of CPU frequency based on the CPU frequency impact and optimality on web request tasks. We can also enable manual control to promote changes based on the application context. The result is a CPU frequency scaling policy for Node.js applications targeting the program's requirements to build sustainable applications while keeping a high standard in runtime performance and thus achieving better resource (energy) utilization. Our goal is to enable DVFS control and provide a method to present and analyze the energy footprint of deterministic actions from within Node.js applications. The main objectives are:

Context-aware DVFS control. Node.js users can adjust the CPU frequency of the processors within the application to execute a specific action and the whole application enabling customizations based on the application requirements. Users do not have to rely on the scaling governors with generic policies. The underlying mechanism for controlling the CPU frequency is based on updating and accessing files in the Linux kernel's *sysfs* filesystem. Thus, minimal configuration and appropriate access rights are required to control the CPU frequency of all cores, making the software easy to use in a production environment.

Energy data collection validity and access. We use the Intel RAPL technology to collect energy data, which has been a research topic of various studies on the validity of its reported data. While there is a difference on actual and reported values, studies show that the data has high accuracy with the plug power. Kashif et al. [2] found that the correlation coefficient of package power with AC power is ≈ 0.99 on a Haswell processor model. However, the DRAM energy deviates more (measurements coincide within 20% [3]) than the CPU energy, while generations following Haswell show higher accuracies for DRAM energy readings [4]. Finally, RAPL has a low performance overhead ($<2.5\%$) [2]. Overall, results deviate on different CPU architectures and DIMM types, while the observed highest inaccuracies happen when the system is idle as well as when the CPUs are idle and the GPU is communicating with the memory [3]. However, studies show

that the technology estimates accurately enough the energy consumed by different domains [2], [3]. To access this energy data, we use the Linux power capping framework [5] and retrieve the energy information though the *sysfs* filesystem as in CPU frequency control.

DVFS-based request-based system. We investigate different request types and characterize them based on the CPU frequency impact on their energy consumption and runtime to create a model of defining and predicting the most suitable CPU frequency on-the-fly. The optimal CPU frequency is determined for each request separately enabling requests to be executed in different CPU frequencies based on their task type.

To this end, we build a system to control the CPU frequency of Node.js (web) applications, we collect data from a variety of action requests representing realistic scenarios and extract patterns on CPU frequency suitability and impact. The contributions are as follows:

1) The *EFreqScaling* module: The Node.js module changes the CPU frequency at which the processor operates, to execute a specified action. It also collects and prints the machine's energy and runtime data for specific operations and the whole application duration.
2) Web-Server CPU Frequency Aware Benchmark Suite. The benchmark suite provides a variety of requests to represent realistic scenarios. It also takes advantage of the *EFreqScaling* module's features to change the CPU frequency of the machine's processors for each request separately and generate its energy/runtime data.
3) Energy Analysis and Modeling. Energy and runtime data are used to classify the requests, present the energy distribution of hardware domains and decide on the most suitable CPU frequency. During this analysis, possible measurements are explored for using the most suitable CPU frequency on-the-fly.

The paper is organized as follows: background and related work in Sections II and III; description of the module and benchmark suite in Section IV; experimental analysis in Section V and conclusion with future work in Section VI.

II. Background

Web Applications are software programs (applications) that utilize the Web browser to present their dynamic content. Their architecture is based on the client/server paradigm, while the programs communicate with each other over the Internet. In this case, the client program is the Web browser that presents the context to the user, while the server(s) process(es) the tasks/requests that arrive from the client side [6].

Many applications, including web applications and web servers, are built with the **Node.js** framework. Node.js is a server-side JavaScript framework with asynchronous and non-blocking I/O, and event-driven logic [7]. The event-based model depends on the concept of the event loop, achieved through the *libuv* [8], [9] library. The library is responsible for the event loop and thread pool functionality. The main (event loop) thread initializes the application and manages all requests and their responses. It maintains a queue of events and executes their callbacks. I/O network operations are executed through non-blocking sockets fulfilled by the event loop, while others, such as the file system, crypto, DNS functions, are executed in a thread pool [8], [10]. By default, four threads, with a maximum of 1024 threads, and a small memory overhead are preallocated and initialized, when a user submits the first work task [11], [12]. Also, Node.js enables the import of any file or organized folder as a module [13] in order to access its functions.

However, as applications utilize the available resources, such as CPUs, they can impact the system's energy consumption. Modern processors can benefit from the **Dynamic Voltage and Frequency Scaling (DVFS)** technique to reduce energy. The mechanism enables CPUs to scale into different frequency-voltage points dynamically to achieve efficiency and power conservation. The reduction of both clock frequency *fclk* and supply voltage *Vdd* decreases the dynamic power, which is a main component of the total power dissipation of CMOS circuits (which are used to build processors): $P_{dynamic} = \alpha C_L V_{dd}^2 f_{clk}$, where C_L is the collective switching capacitance and α the switching activity. The equation estimates the dynamic power consumption of CMOS circuits and reveals the quadratic relation between supply voltage and power and the linear relation between switching frequency and power. A reduction of the supply voltage causes circuit delays, which in turn limits the operating frequency [14]. Thus, to conserve power and function properly, both frequency and voltage points need to scale accordingly.

These processors operate in a set of various frequency and voltage configuration points (Operating Performance Points or P-states). In the Linux kernel, the CPU Frequency scaling (CPUFreq) subsystem consists of the core (basic framework), scaling governors and scaling drivers to support the CPU performance scaling. The governors implement scaling algorithms that estimate the CPU capacity and can be potentially parameterized, while the drivers communicate with the hardware and provide the available p-states to governors. Also, drivers change the CPU p-states as instructed by the governors. Some generic scaling governors are: "performance", "powersave", "userspace", "schedutil" and "ondemand". The *sysfs* directory holds the policy interface (*/sys/devices/system/cpu/cpufreq*) for each CPU where various attributes are exposed and can be configured to control the governors' algorithms. The most fine-tuned algorithm comes from the "userspace" governor. In this case, the governor allows the user space to define the CPU frequency by writing in the policy's *scaling_setspeed* attribute [15].

III. Related Work

Predictive DVFS-based Systems. Dhiman et al. [16] implemented a system level DVFS technique in which they characterized tasks and determined memory-bound and CPU-bound phases using the Cycles Per Instruction (CPI) metric. Their online learning framework determined the most suitable frequency-voltage setting and aimed for memory-bound phases

to run on low frequencies, CPU-bound phases on high frequencies and combinations on intermediate frequencies using cost functions. Choi et al. [17] built the DVFS-WD regression-based technique for non-realtime applications that was tied to the Linux OS scheduler. The policy used a performance monitoring unit (PMU) to decompose the task workload and determine the on-chip and off-chip latencies, which were used to decide on the appropriate frequency. A hardware-based solution was used to measure energy consumption. Rountree et al. [18] considered the Leading Loads architectural model, which measured the leading (first) memory reads that caused cache misses. They measured them with their Leading Loads Cycles hardware performance monitor (HPM), which estimated the bus time. The HPM was used to predict the execution time on frequencies in DVFS. Jongmoo Choi et al. [19] built a low-power framework for multicore mobile devices that incorporated DVFS and DPM (Dynamic Power Management) techniques. Their parametrized system increased/decreased the CPU frequency (within minimum and maximum values) and activated/deactivated heterogeneous cores based on CPU resource utilization thresholds. Their framework used an online resource usage monitor component to provide information about various resource statistics, using the proc file system, and report the data on a web page using Node.js. Acun et al. [20] used a fine-grained frequency regulation approach to apply optimal frequencies at the function level within applications using the per-core DVFS approach. They implemented their runtime module in the CHARM++ framework and collected statistics data on power and performance for different frequencies of each entry method in every charm instance in the framework. Then, they calculated the energy-optimal frequency based on the data and they applied the appropriate frequency for each method, taking into account frequency transition latency and overhead. In contrast, our study focuses on characterizing and analyzing whole web requests using energy and runtime metrics to expose patterns and the frequency impact and optimality measured in a machine with per-chip DVFS support. We do not use, modify or add any hardware counters directly, instead our solution is based on information and configuration that is available through user interfaces.

Application Energy Analysis & Optimization. Li et al. studied the energy consumption of HTTP requests for mobile applications [21]. In their approach, they optimized sequential requests by bundling them together in a single request using a Node.js proxy server. Lago and Larizgoitia [22] built an application-aware framework that used the DPM techniques on the network cards of mobile devices. They reduced the power consumption, by launching the greatest number of pending requests when a new network connection was made to increase the idle periods with their CIST (Consumption Improvement by Stuffing Time) technique. In idle periods, the wireless network card was set to operate on low (idle) mode to save energy. Cui et al. [23] identified the bottlenecks of tail latency of Node.js web services using the Event Dependency Graph (EDF) to represent event relationships of every request. They increased the CPU frequency to the maximum value when V8's GC operations were triggered and when the event loop queue became busy to optimize the request performance with little energy overhead. Beyer et al. [24] built the CPU Energy Meter tool that measured the system's energy using the Intel Running Average Power Limit (RAPL) energy consumption feature. It was integrated into the BenchExec benchmarking framework to reveal energy data of verification algorithms. Liu et al. [25] created the Chameleon system that was implemented in the Linux OS kernel to reduce energy consumption of mobile processors. The prototype added system calls, expanded the /proc and /dev interfaces, process control block properties and the Linux scheduler. They changed the frequency and voltage settings by writing the values in two machine-special registers (MSRs). They estimated the future processor demand by using past CPU usage statistics and used the estimated processor availability to determine the processor speed. In contrast, we aim to optimize Node.js web requests with CPU DVFS, by exploring available CPU frequencies that can provide a balance between execution time and energy consumption. Energy information comes from the Power Capping Framework with Intel Running Average Power Limit (RAPL) support, which requires accessing files, thus a minimum configuration is needed to read the energy consumption from our tool in a production environment. Finally, our approach enables CPU DVFS control from the application level to allow developers to change the frequency based on the context.

IV. CPU Frequency Scaling/Energy Aware System

The *EFreqScaling* module enables the CPU frequency control within Node.js applications using the DVFS technique. It also exposes energy and runtime information for specified actions. The module is imported in the benchmark suite to control the CPU frequency of the machine's processors to execute web requests and measure their performance.

A. EFreqScaling Module

The module consists of a "settings.js" file and the source code files. The settings file defines various parameters including the *scaling_setspeed* paths for setting the frequency of all CPUs, the available frequencies sorted in descending order, the default frequency, the paths of domains' energy data and other parameters related to the monitor thread (which is not used and it is currently deactivated for this work). The main operations are: functions that change the CPU frequency of all CPUs and functions that measure the energy and runtime performance for a task. Every operation has a different API function based on the task structure type: asynchronous (with callback), synchronous and event-based actions. Their internal mechanism is described below:

- **CPU frequency control APIs**. We change the frequency of all CPUs by writing the frequency value in the files located in the CPU Frequency scaling (CPUFreq) subsystem (*"/sys/devices/system/cpu/cpuY/cpufreq/"*). The CPU frequency is set back to a default value after the task

completion. Currently, there are three ways for setting CPU frequency: lowPriority that sets the CPU frequency at a minimum value, highPriority that sets the CPU frequency at a maximum value and the current state as critical for the duration of the function and reqPriority that sets the CPU frequency at a requested value. The functions handle log files to determine whether a CPU frequency can change, e.g. current request is not high priority, but current frequency remains at maximum due to concurrent, yet-to-be-completed, high priority functions. The highPriority and lowPriority prototypes are built for synchronous and asynchronous tasks, that can be used to directly adjust the CPU frequency of processors to execute actions that are considered high and low priority from the context. The reqPriority includes APIs for asynchronous, synchronous and event-based actions. For the first two cases, we build function wrappers and for the third case we create separate functions that change the frequency to a requested value and back to the default value. Overall, the process of updating files for all CPUs creates minor overhead from the Node.js level causing up to 3ms (for a machine with 12 CPUs) to be spent to write in the files of all CPUs for every frequency change.

- **Energy/runtime APIs**. We measure and report the energy of package, core, rest-uncore and DRAM domains. Energy consumption is calculated using the Linux power capping framework [5] with the Intel Running Average Power Limit (RAPL) technology. Intel RAPL measures the energy consumed for specific domains x with the MSR (model-specific registers) interfaces: *MSR_domain$_x$_ENERGY_STATUS*. The registers of PKG (processor die), DRAM, PP0 (processor cores) and PP1 (uncore graphic device for client platforms, if supported) domains are updated every $\sim$1msec [26]. Our APIs read the *energy_uj* counter (current energy expressed in microJoules) from package, DRAM and cores domains in the beginning and at the end of a task by accessing the corresponding files in the *powercap* interface (*"/sys/class/powercap/"*) of the supported power domains. The energy spent in Joules (J) for package, core and DRAM devices is calculated with the following formula: $\frac{(energy\_domain_i\_after - energy\_domain_i\_before)}{1,000,000}$. In case of the rest-uncore devices, we calculate the consumption using the data from package and core domains, based on this relationship: $core + uncore <= package$ [27] on energy consumption. By subtracting the energy of core from the package domain, the rest-uncore energy should also include the energy consumed from the last level caches and memory controller [28]. Finally, the execution time is measured with Node.js's *new Date()* method. As in the CPU frequency control APIs, there are three ways to calculate energy for a task: two are implemented as function wrappers for functions, with and without callbacks, and separate functions built to be called at the beginning and at the end of a task.

B. Web-Server CPU Frequency Aware Benchmark Suite

The benchmark simulates a real-world Node.js web application. It is necessary to build a benchmark that utilizes technologies, modules, architectural features and tasks-scenarios found in industry to determine the CPU frequency impact on web applications. We use popular components and construct a benchmark suite as an example of common Node.js web applications. We include simple tasks, such as iterative calculations, and more complex ones, such as database calls and file parsing to investigate the CPU frequency suitability on different scenarios in a controlled web application set-up. We examine cases that vary the complexity-duration of tasks to decide on the workload variance on the CPU frequency suitability.

The web application is constructed with the Express [29] engine. Express is a Node.js web framework with features for web applications. Using the engine's mechanisms, we create a small web server following the Model-View-Controller [30] web presentation pattern. The application uses the EJS [31] template engine for processing the input data and templates and generating the HTML pages. It also uses the MongoDB [32] document-based database that stores the records with field-value structure, called documents, which are equivalent to JSON objects. The documents are stored in collections, which are similar to tables in relational databases [33]. Finally, the Mongoose [34] object modeling library is used to manage the application data that are stored in the MongoDB database.

The *SampleCollections* database, used by requests that perform DB queries, is imported from the dbkoda-data [35] github repository. Also, JSON files from Hawaii Open Data: Unemployment Data [36] and Statewide Unemployment [37] are used and modified to increase the rows and perform disk-intensive functions. Our modified unemployment file sizes are: 866.9KB and 20.1KB respectively.

During startup, the application connects to the MongoDB database and the Mongoose services. The requests are named based on the hardware and software components that are meant to use/stress the most or describe an action and whether they perform asynchronous functions that involve using callbacks:

- **requestCpuFunction**. The request iterates through n numbers (n given as a request parameter) and sums numbers from 0 to n. When it finishes the task, it returns the result back to the client.
- **requestCustomersFunctionAsync**. The request performs a DB query with a Mongoose Object wrapper. It returns all the customers (100,000 records) sorts them by name and selects their name and address field to appear for each record.
- **requestDiskFunctionAsync**. It reads the two JSON Unemployment files and joins their data rows based on year. It selects 500 rows and creates 500 records in the database (UnemploymentCodeJoin collection) from the joined rows.

- **requestArrayFunction**. The request iterates through n numbers and creates data with the format: $\{$"id" : i, "sum" : sum, "avg" : $\frac{sum}{i}\}$ from 0 to n and pushes them to an array. When it finishes the task, it returns the array back to the client. The request is a variation of requestCpuFunction, but it creates new memory data in every iteration.
- **requestMemoryFunctionAsync**. It calls the requestDiskFunction request from another machine to receive data. The requestDiskFunction reads two files, joins and sorts their data rows based on year and returns the joined rows as strings (JSON.stringify). The requestMemoryFunctionAsync receives the rows, parses and sorts them based on their id and sends them back to the client.
- **requestOrdersFunctionAsync**. The request finds the 100 most and the 100 least purchased items from the database. It aggregates and groups by the Order collection to find the purchased (4,853) products. Afterwards, it populates their fields to collect all the data from the Product fields. The most and least purchased ones are retrieved and sent back to the client.
- **requestReceiveFunctionAsync**. It calls another request from another machine and returns immediately the request's response back to the client. It calls the requestCpuFunction request with n number of iterations, which is passed from its parameter.
- **requestSocketFunction**. The function simulates a conversation between server and client (web browser) that use web sockets with limited computation. During startup the application initializes the socket listeners. When the request is called from a web browser, a chat communication is initiated between the server and the client using these listeners. A predefined "conversation" occurs with chat messages exchanged between them through web sockets. The responses are sent with a time delay based on the character count of each response (50ms per character) to represent a near-realistic user response time. The responses are shown on the client web page. When the conversation ends, the client user is redirected to the home page to simulate the end of their discussion.
- **requestCryptoFunctionAsync**. The request uses the crypto.pbkdf2 function that derives a password-based key. A number is passed as request parameter and used in the function to derive a secure key. The function uses the libuv's thread pool internally [38].
- **requestWorkerThreadFunctionAsync**. It creates w Worker Threads and sends them an iteration flag. In case the flag is zero, the worker threads read two files (similar to requestDiskFunctionAsync), join and sort the rows based on the year. Each worker returns 50 rows to the main thread. In this case (0) the main thread saves all the rows received from the thread in the UnemploymentCodeJoin collection. In case the iterations parameter is greater than zero (hw), the worker threads perform a loop and create data in every iteration with the format $\{$"id" : i, "sum" : sum, "avg" : $\frac{sum}{i}\}$ from 0 to n (similar to requestArrayFunction). The arrays are sent back to the main thread. The main thread collects them and sends them back to the client. The workers are spawned for every request.
- **requestWorkerThreadAliveFunctionAsync**. The function is similar to requestWorkerThreadFunctionAsync. However, w Worker Threads are initialized once in the beginning of the web application. In the request, the main thread's callback is defined. When the request is triggered, the main thread sends a message to the threads to start the task, removes any previous attached listeners from the threads and attaches the above callback to be triggered on workers' message emission event. The tasks executed by the workers are the same as described in requestWorkerThreadFunctionAsync.

When the requests complete, they either render the data on the client side using a template file to present/organize the information or they return the data as an HTTP response without a template involved. Besides the individual parameters, every request accepts a frequency value f as a request parameter. The URL format, using the requestSocketFunction request as an example, is: *"<domain_name or IP:port>/socket/start/:freq"*.

The requests use the EFreqScaling module's CPU frequency control APIs to change the current CPU frequency of the cores to the requested f for the duration of their actions. They, also use the energy/runtime APIs to measure the energy consumed and the runtime of the whole process. The energy/runtime wrapper APIs include the overhead of transitioning to the requested frequency from a default one and back, while in the chat request we start measuring the energy/runtime after the CPU frequency is set to the requested values.

C. Experimental Design & Metrics

We start the web-server (benchmark suite application) and the application sets as a default the lowest available frequency of the machine (800MHz). We collect data for every request. We vary the iterations of specific requests to represent low (lw) and high (hw) workloads and the number of workers (w) spawned. In every scenario, the requests run 100 times for every available frequency f and the sequence is repeated three times. All 300 request-data-per-frequency values are gathered and aggregated to produce their average and standard deviations:

- Energy. We collect the energy consumption of package (energy-Package), core (energy-Core), rest-uncore (energy-Rest-Uncore) and DRAM (energy-DRAM) domains for every case. We calculate the total energy spent by summing together the data from package and DRAM domains.
- Runtime. We use the execution time as reported by the EFreqScaling module.

1) Request "Best" Performance Metric: DVFS enables the utilization of various frequencies affecting the energy/runtime.

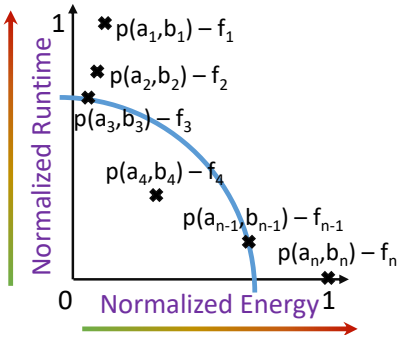

Fig. 1: Euclidean distance of normalized energy/runtime metrics

To find the CPU frequency that enables the most efficient balance of energy consumption and runtime, we need to define a "best" performance metric. To this end, the "best" is defined in terms of execution time and total energy consumption (package and DRAM energy). High CPU frequencies usually cause request tasks to run fast at the expense of high energy consumption. Both metrics are considered equal factors in finding the CPU frequencies that provide a fine balance between fast runtime and low energy usage. To this end, we use the average total energy and execution time for every frequency and request type, as described above.

The process is transformed to a non-linear multiobjective (Pareto) optimization problem [39] in which energy and runtime values are the two objective vectors that are minimized simultaneously—they are equally important—for a defined set of decision vectors representing CPU frequencies. Pareto optimality in a multiobject scenario describes the concept of optimization as a set of solutions, each one with components that cannot be optimized further without deteriorating any other component [39]. In our scenario, an optimal CPU frequency does not enable optimal runtime and optimal energy consumption, but an optimal combination of these metrics.

First, we normalize the energy/runtime metrics using the Feature Scaling Normalization: $x' = \frac{x - x_{min}}{x_{max} - x_{min}}$, where x_{max} and x_{min}, are the min and max values for each request type and metric. At this point for every request type and frequency, the normalized total energy and normalized execution time have numbers that vary between zero and one. Given a plot for a request type, for every frequency f there is a point *p(a,b)* with coordinate *a* being the n_{energy} point and coordinate *b* being the $n_{runtime}$ point (Figure 1). The farther (larger) the point is from the beginning of each axis, the more inefficient the frequency it represents is (higher execution time and/or energy). To find the most efficient combination, we calculate the distance from (0,0) point with this formula using the Euclidean distance metric = $\sqrt{n_{energy}^2 + n_{runtime}^2}$. The higher the number the farther a point is from $(0, 0)$, thus the more inefficient the frequency for this task is. Lower numbers are more efficient.

The approach is based on the method of Global Criterion with Euclidean distance (distance metric) calculated from the (0,0) reference point to the normalized energy and runtime *(a,b)* pairs. The method provides a Pareto optimal solution [39]; in our case, this means a solution where the (scaled) energy consumption cannot be decreased without increasing the (scaled) runtime.

V. Experimental Analysis

Tables I and II depict the requests that perform the corresponding request tasks described in Section IV-B. The requests have a variety of iterations (low workloads–lw and high workloads–hw), shown in the second column of the tables, and number of workers to explore in the different cases, presented in parentheses as part of the request name.

A. Environment-Setup

We ran the experiments with Node.js version 14.4.0 on a desktop machine under Ubuntu 16.04 LTS, with 32GB RAM, six Intel i7-8700 cores and two hardware threads per core. It supports DVFS with processor base and maximum frequencies: 3.2GHz and 4.6GHz [40]. We set the "userspace" governor with ACPI CPUfreq (*acpi-cpufreq*) scaling driver. The available frequency values, as reported in the *cpupower* tool, with this driver are: 800MHz, 1000MHz, 1.1GHz, 1.3GHz, 1.5GHz, 1.7GHz, 1.8GHz, 2GHz, 2.2GHz, 2.3GHz, 2.5GHz, 2.7GHz, 2.9GHz, 3GHz, 3.2GHz and the turbo boost maximum frequency. The turbo boost maximum frequencies are: 4.6GHz when one core is active, 4.5GHz when two cores are active, 4.4GHz when three cores are active and 4.3GHz when four to six cores are active [41].

We also use an ASUS VivoBook laptop under Arch Linux OS, with 16GB RAM, four Intel i7-8550U cores and two hardware threads per core and DVFS support. The machine is used to run the scripts for sending the requests to the desktop machine (where the benchmark suite runs), receive the responses, run the benchmark suite for requests that require communication with other machines and executing the client side (web browser) for the chat communication scenario.

B. Request Groups

First, we evaluate the performance of every request and frequency. Tables I and II show the minimum and maximum values of (total) energy consumption and runtime metrics based on the aggregated average values per frequency. The tables also reveal the effect of CPU frequency on the runtime and energy consumption of each request. The requests in Table I show a bigger effect than the requests in Table II. Based on the runtime percentage difference ($|\frac{Max - Min}{\frac{Max+Min}{2}}| * 100$) between minimum and maximum values at the 8^{th} column, we split the requests into two groups.

The groups show whether the request performance depends on the CPU frequency of the machine (CPU-driven requests in Table I) or the CPU is not the main/sole computer component that they use to complete a task (Non/Less-CPU-driven requests in Table II). We define as CPU-driven the requests

Request	Iterations	Min Runtime (msec)	Max Runtime (msec)	Min Energy (J)	Max Energy (J)	Max estim. Freq. (MHz)	Runtime %diff	Energy %diff
Cpu_lw	8,000,000	60.97	283.02	0.76	1.40	4,600	129.10	58.90
Cpu_hw	16,000,000	646.48	2,308.72	8.36	18.63	4,600	112.50	76.08
CustomersAsync	N/A	1,075.13	3,775.04	17.23	37.09	4,600	111.33	73.12
Array_lw	50,000	132.90	622.48	1.87	3.80	4,600	129.62	67.83
Array_hw	100,000	289.44	1,264.01	4.08	8.53	4,600	125.47	70.52
OrdersAsync	N/A	107.38	383.00	1.51	2.87	4,600	112.41	62.28
DiskAsync	N/A	98.87	460.15	1.55	2.94	4,600	129.26	61.93
CryptoAsync_lw	200,000	112.34	503.10	1.38	3.01	4,600	126.99	74.07
CryptoAsync_hw	2,000,000	1,051.46	3,626.60	13.86	31.72	4,600	110.09	78.35
WorkerThreadAsync_0 (1 W)	0	46.40	197.42	0.62	1.04	4,600	123.87	51.09
WorkerThreadAsync_0 (3 W)	0	70.70	305.12	1.35	2.53	4,500	124.75	60.87
WorkerThreadAsync_hw (1 W)	100,000	145.57	594.06	1.98	4.10	4,600	121.28	69.58
WorkerThreadAsync_hw (3 W)	100,000	324.97	1,321.27	5.34	11.78	4,500	121.04	75.27
WorkerThreadAsync_hw (6 W)	100,000	645.69	2,532.02	11.91	25.53	4,400	118.72	72.72
WorkerThreadAsync_hw (12 W)	100,000	1,205.52	4,440.61	23.01	49.92	4,300	114.59	73.78
WorkerThreadAliveAsync_hw (1 W)	100,000	113.83	455.92	1.62	3.27	4,600	120.08	67.63
WorkerThreadAliveAsync_hw (3 W)	100,000	282.64	1,170.30	4.53	9.94	4,500	122.19	74.68

TABLE I: CPU-driven Requests - Min/Max values from Average values of Runtimes and Total Energy per Frequency

Request	Iterations	Min Runtime (msec)	Max Runtime (msec)	Min Energy (J)	Max Energy (J)	Max estim. Freq. (MHz)	Runtime %diff	Energy %diff
MemoryAsync	N/A	135.98	155.86	0.33	0.42	4,600	13.62	24.74
ReceiveAsync_lw	8,000,000	108.85	116.53	0.25	0.34	4,600	6.82	30.43
ReceiveAsync_hw	80,000,000	4,676.52	4,761.74	7.86	8.74	4,600	1.81	10.68
SocketChat	N/A	25,291.70	25,631.54	41.47	43.34	4,600	1.33	4.41

TABLE II: Non/Less-CPU-driven Requests - Min/Max values from Average values of Runtimes and Total Energy per Frequency

that have more than 85% runtime percentage difference (for example min=1,000msec and max=2,500msec), while requests with lower values are considered non/less-CPU-driven.

The CPU-driven requests in Table I have runtime percentage differences more than 110% and energy percentage differences more than 58%. The requests include (with any number of iterations): tasks that process data in a loop manner; database operations; usage of libuv's thread pool to execute an encryption algorithm and access the file system; and usage of worker threads to perform iterative and file-parsing tasks in parallel. The tasks are mostly CPU-intensive involving database, disk and multi-thread (multi-CPU) activity.

On the other hand, the non/less-CPU-driven requests in Table II have less than 14% and 31% percentage differences between minimum and maximum values in runtime and energy metrics respectively. Here, the tasks involve request calls to another machine and communication with websockets. Thus, the tasks are more I/O intensive than the CPU-driven ones.

In the same tables, there is the estimated maximum (turbo boost) frequency (MHz) upper bound (7^{th} column) for every case based on the CPU hardware specifications and the user threads. In cases where multiple threads are spawned, the parallel part of the request has a reduced estimated maximum frequency.

C. Energy Domain Distribution

Figure 2 presents the energy consumption of each power domain and the runtime performance for the Cpu_hw request. The task has close to 100% CPU utilization for every set frequency measured and it is previously categorized as CPU-driven. Since the task utilizes CPU cycles greatly, its performance and energy is affected by the CPU frequency and voltage values with performance and (CPU) energy consumption increasing as CPU frequency and voltage increase. The graph shows how the total energy (stacked columns per set frequency) changes as the set CPU frequency of the processors scales. For set frequencies between 800MHz and 1300MHz, the total energy starts with ≈ 8.91J and does not decrease lower than ≈ 8.89J (at 1300MHz). For frequencies higher than 1300MHz, the energy consumption slightly drops starting at 1500MHz reaching the lowest point at 1800MHz (≈ 8.36J) and then it increases steadily as CPU frequency (and corresponding voltage) scale up.

Among the energy-Core, energy-Rest-Uncore and energy-DRAM, the energy-Core follows the same trend as the total energy and has the highest values among the domains making the cores the dominant energy consumers for this request. Energy-DRAM and energy-Rest-Uncore decrease as CPU frequency increases. As previous studies indicated, components such as memory and I/O interfaces may remain active and consume power throughout the duration of a task [42]. By lengthening a CPU-bound task, the memory remains active in a standby state longer, performing for example memory refresh operations, and thus, it increases its energy consumption for lower frequencies [42], [43]. Also, runtime (line in the graph) is decreased as the set CPU frequency increases. Overall, runtime

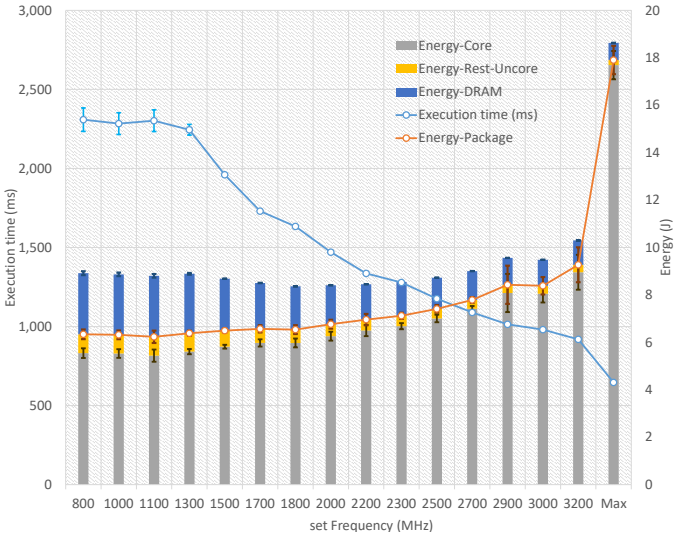

Fig. 2: Cpu_hw domain energy distribution and runtime per frequency

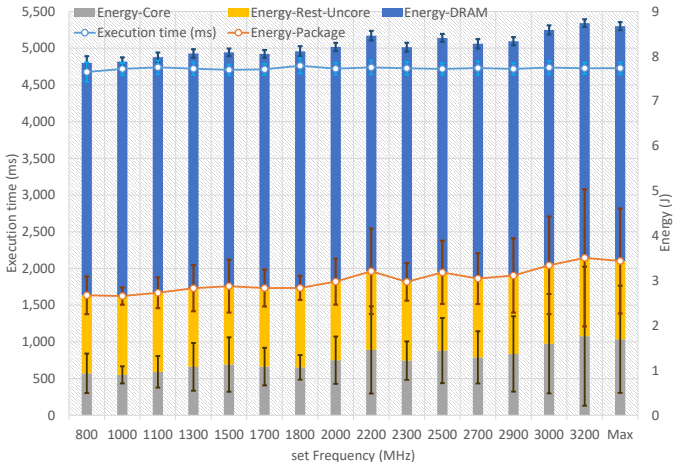

Fig. 3: ReceiveAsync_hw domain energy distribution and runtime per frequency

has a higher slope as CPU frequency increases compared to total energy consumption in intermediate frequencies, where energy-Core increases steadily and energy-Rest-Uncore and energy-DRAM decrease steadily with lower rates.

Figure 3 presents the energy consumption of each power domain and the runtime performance for the non/less-CPU-driven ReceiveAsync_hw request. The task is I/O-intensive and it has less than 1% CPU utilization for every set frequency measured, making the CPU frequency a non-dominant factor for affecting its performance and energy consumption. The energy-DRAM has the highest value followed by the energy-Rest-Uncore and energy-Core. The energy-Core increases as CPU frequency increases with some minor drops in a few cases. Energy-DRAM and energy-Rest-Uncore values are almost steady at ≈ 5.2J and ≈ 1.7J respectively and higher than in the CPU_hw request. In this case, the main domain energy consumer is memory. However, the energy of each domain separately and the total energy have a very small difference with a minor increase as frequencies scale up. Also, there is a small change in the execution time with a slightly larger standard deviation than the CPU-driven request above. Overall, the request has a steadier trend in energy and runtime as set CPU frequency scales than the CPU-driven task indicating that it is susceptible to factors other than CPU frequency, such as network latency, to affect its performance.

D. Request Best Performance Investigation

Figures 4 and 5 present the distances of normalized runtimes and total energy for each type as described in Section IV-C1. The CPU-driven requests in Figure 4 show a very consistent pattern for all requests. More specifically, the distances of the requests are close to one at the lowest (800MHz) frequency, at 1000MHz and 1100MHz in some cases (where the highest runtime was observed), and at the highest (Max) frequency (where the highest energy was observed), showing the worst overall performance compared to the rest frequencies. As frequencies increase between 1300-3200MHz, the distances reduce until they reach a local minimum (Pareto optimal point) and then they increase and reach the high distance value of the turbo boost frequency. Such a pattern shows a direct and high effect of the CPU frequency on every request.

In the same figure, we calculate the 40^{th} percentile (0.355) using the data from all the requests/frequencies from this group. This shows that the distance value 0.355 is higher than 40% of all the distances and lower than 60% of all the distances. The percentile provides a boundary across the distance values in order to determine the ones that are better than the rest. The percentile is chosen to be higher than the local minimal (best) distance for every request type (the maximum local minimal is 0.344). Any distance below this line is considered to give a good runtime/energy consumption performance for the corresponding CPU frequency, while any distance above the line is considered to be too large and thus the frequency is unsuitable for the request to run. Based on this, there is at least one request that is below the 40^{th} percentile value between 1800-3200MHz, while at 2700MHz and 2900MHz all requests have lower distances than the percentile. Based on the above, the 2700MHz and 2900MHz show the best performance for this group.

In Figure 5, the non/less-CPU driven requests show dispersed distance values across the different frequencies. The distances are spread in higher and lower peaks than the first group without a very clear trend in most cases. The SocketChat shows a clearer trend. In all cases, high frequencies show worse performance than the rest. Here, the 10^{th} percentile (0.272) is used as a boundary limit in which the local minimal distances for every request are below it (the maximum local minimal is 0.270). There is no frequency at which every request has a distance value lower than this percentile, but in frequencies including 1500MHz and 1800MHz and above, all distances are higher than 0.278. Thus, frequencies 1500MHz and equal or higher than 1800MHz are considered unsuitable for this group. However, there is at least one request that is

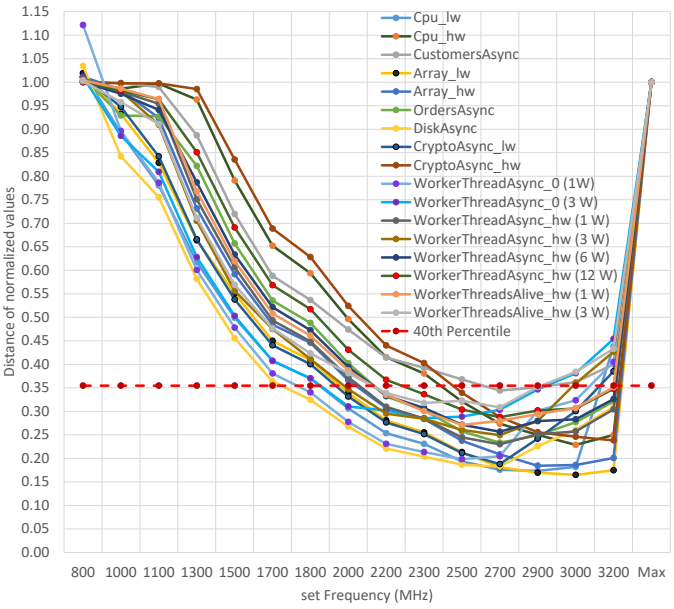

Fig. 4: CPU-driven Requests — Distances of Normalized Total Energy and Runtime

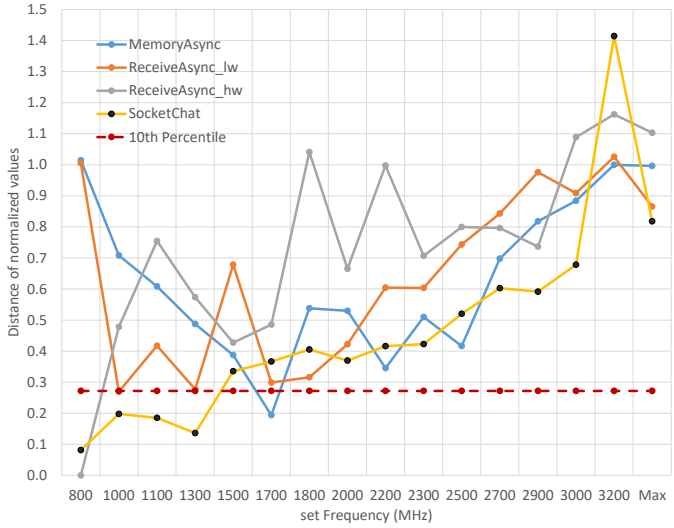

Fig. 5: Non/Less-CPU-driven Requests — Distances of Normalized Total Energy and Runtime

below the percentile in frequencies 1700MHz and 1300MHz and lower, making them more appropriate for this group.

E. Request Group CPU Frequency Impact

Based on the most efficient CPU frequencies of each group, we calculate the average runtime and energy percentage changes from a suitable frequency to reveal the performance impact when tasks are executed in a non/less suitable frequency. First, we find the percentage change $(\frac{Value_{fi} - Value_{suitfreq}}{Value_{fi}} * 100)$ in runtime and energy of each CPU frequency from the most suitable one for each request and then we calculate the average for the group.

Frequency (MHz)	Average Runtime %Change	Average Energy %Change
800	160.04	-1.34
1000	147.71	-4.56
1100	137.02	-6.05
1300	106.83	-9.32
1500	79.44	-11.02
1700	58.55	-11.31
1800	49.36	-10.80
2000	34.69	-8.87
2200	22.43	-6.58
2300	17.18	-6.17
2500	7.88	-3.29
2700	0.00	0.00
2900	-6.80	5.68
3000	-9.51	8.57
3200	-14.97	15.56
Max	-36.39	79.71

TABLE III: CPU-driven Requests — Runtime and Total Energy Percentage Change Averages from 2700MHz

Frequency (MHz)	Average Runtime %Change	Average Energy %Change
800	2.96	-0.86
1000	0.94	-2.68
1100	0.97	-1.86
1300	0.00	0.00
1500	0.40	0.89
1700	-1.06	0.69
1800	-0.04	2.73
2000	-0.19	3.93
2200	-1.07	6.29
2300	-0.28	5.76
2500	-1.34	8.29
2700	0.63	7.56
2900	0.32	10.17
3000	0.10	12.65
3200	-1.22	16.28
Max	-1.31	14.61

TABLE IV: Non/Less-CPU-driven Requests — Runtime and Total Energy Percentage Change Averages from 1300MHz

In Table III, the energy/runtime impact of the CPU-driven requests from 2700MHz is presented. The 2900MHz frequency has a very similar increase in energy consumption (5.68%) with runtime deduction of 6.8% in comparison to the 2700MHz. While, for the lowest frequencies, the group experiences slowdowns more than 100% than the 2700MHz case and in the highest frequency there is an increase of $\approx 80\%$ in energy consumption. The group shows a severe impact as frequencies move farther away from 2700MHz.

In Table IV the performance impact of non/less-CPU-driven requests from 1300MHz is presented. The CPU frequency is one of the highest ones (1700MHz is the highest) in which there is at least one request with distance lower than the 10^{th} percentile value described above. The percentage changes from 1300MHz in both metrics are very little in comparison to the CPU-driven requests revealing a wide range of frequencies, especially 1100MHz-1700MHz, that can be also used for running the requests of this group with little runtime/energy %change. However, there is a steady increase in energy consumption with no substantial runtime increase as CPU frequency increases to the maximum value.

The 1700MHz frequency shows a decrease of 1.06% in execution time with 0.69% additional energy consumption in comparison to the 1300MHz case. Thus, we can either choose 1300MHz frequency to run non/less-CPU-driven requests or the 1700MHz with minimal runtime increase, but higher energy consumption.

F. Request Group and Best Frequency Model

Lastly, we investigate whether we can determine the request group and then apply the appropriate frequency on-the-fly. We use the ratio between total energy over execution time from the above data to find the energy per second (J/s). The idea relies on the fact that when a request arrives, we can measure the total energy consumption and runtime, calculate the ratio and approximate the energy per second for a task. The ratio, also describes the power (Watt=Joules/second) used in a given time to fulfill a task.

In Figure 6 the energy per second for each CPU-driven request shows a steady increase as frequencies increase. We assume a linear dependency of energy consumption on CPU frequency and that the ratio between energy and runtime can indicate whether the CPU frequency provides optimality. Thus, we find the ratio of every request and frequency and we extract the minimum and maximum ratios in which the CPU frequency they run on gives the best performance. The minimum and maximum ratios are shown with lines (MinT and MaxT) for 2700 and 2900MHz in which the best performance is observed based on our previous analysis on distance of normalized values in Figure 4. Using these lines we calculate the linear trendlines for both min and max boundaries using Excel's Trendline tool. Any CPU-driven request that has energy per second between these boundaries should have high performance for the corresponding frequency. The positive y's of the lines are stretched between 1500MHz and Max frequencies. The disadvantage of this model is that it can include frequencies that are not within the optimal space defined by the normalized distances and/or exclude alternative optimal ones. However, the trends can include different frequencies for every request, besides the two calculated ones, if the energy per second falls between these boundaries.

In Figure 7, the energy per second for non/less-CPU-driven requests does not show as high increase as above. The values increase minimally as frequencies increase. Slightly high standard deviations are observed, while the values do not scale. The requests show low energy per second consumption, indicating being "greener" than the group above. Their average ratios are constantly lower than the ones from the CPU-driven requests with their averages between 1.638 to 3.114J/s. More specifically, at 1300MHz frequency the minimum average ratio of CPU-driven requests is 3.895J/s, while the maximum average ratio for non/less-CPU-driven requests is 2.484J/s. At 2200MHz, the values are 6.325 and 2.834J/s respectively, increasing the gap between these groups and making high CPU frequencies more suitable to determine the request group accurately using a threshold. Thus, the ratio can be used to determine whether a request belongs to either group.

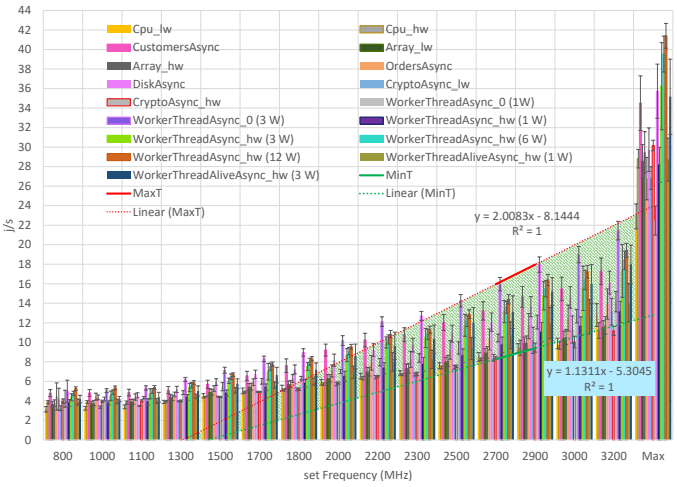

Fig. 6: CPU-driven Requests — Total Energy Runtime Ratio

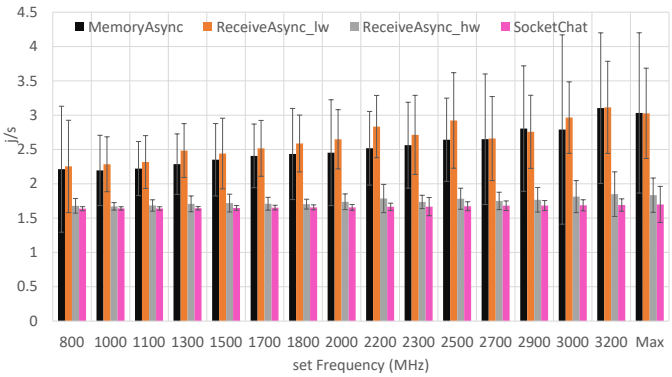

Fig. 7: Non/Less-CPU-driven Requests — Total Energy Runtime Ratio

VI. Conclusion & Future Work

In this study, we used the CPU DVFS technique to enable web applications to tune CPU frequencies accordingly. To this end, we built a CPU frequency/energy aware benchmark suite and module that exposed the energy consumption of various power domains. Our system measured the energy consumption/runtime of various web requests executed sequentially on the available frequencies to extrapolate data on the CPU frequency optimality for each request separately.

We categorized the requests into: CPU-driven and non/less-CPU driven, based on the CPU frequency impact on them. We considered energy and runtime equal factors to determine optimal CPU frequency/ies and identified them for each group. CPU-driven requests had a clear trend on CPU frequency "suitability" with 2.7GHz and 2.9GHz the most efficient ones for this machine, while they were the ones that had the most severe impact when moving away from the optimal CPU frequency. The non/less-CPU-driven ones did not have a clear trend as CPU frequency scaled, had a smaller CPU frequency impact and consumed less energy. Suitable frequencies for this group and machine were: 1.3GHz and 1.7GHz, while higher frequencies were ineffective for this group. Although, every

request in both groups did not have the same optimal CPU frequency, our results showed that we can define an efficient CPU frequency for each group separately transforming the problem into identifying the request group correctly and fast. The ratio for energy/runtime showed to be a good candidate for determining the request category on-the-fly, using a threshold.

In the future, we will incorporate the above knowledge and build a CPU frequency scaling policy for Node.js web applications. Our analysis shows that, we can fine-tune the CPU frequency on-the-fly for each request based on the group they belong to achieve efficiency.

ACKNOWLEDGMENTS

This research was conducted within the Centre for Advanced Studies—Atlantic, Faculty of Computer Science, University of New Brunswick. The authors are grateful for the colleagues and facilities of CAS Atlantic in supporting our research. The authors would like to acknowledge the funding support of the Natural Sciences and Engineering Research Council of Canada (NSERC), 501197-16. Furthermore, we would also like to thank the New Brunswick Innovation Foundation for contributing to this project. We also thank Stephen MacKay for his careful proofreading and editing the paper to improve its quality.

REFERENCES

[1] L. A. Barroso, U. Hölzle, and P. Ranganathan, "The datacenter as a computer: Designing warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, vol. 13, no. 3, pp. i–189, 2018.

[2] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 3, no. 2, pp. 1–26, 2018.

[3] S. Desrochers, C. Paradis, and V. M. Weaver, "A validation of dram rapl power measurements," in *Proceedings of the Second International Symposium on Memory Systems*, 2016, pp. 455–470.

[4] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the intel haswell processor," in *2015 IEEE international parallel and distributed processing symposium workshop*. IEEE, 2015, pp. 896–904.

[5] T. kernel development community, "Power Capping Framework," https://www.kernel.org/doc/html/latest/power/powercap/powercap.html, 2021, [Online; accessed 29-January-2021].

[6] L. Shklar and R. Rosen, *Web Application Architecture Principles, Protocols and Practices*. John Wiley & Sons, Inc., 2003.

[7] S. Tilkov and S. Vinoski, "Node. js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[8] "Libuv 1.20.4-dev Documentation," http://docs.libuv.org/en/v1.x/design.html, 2014, [Online; accessed 13-July-2018].

[9] L. contributors, "Libuv Documentation Release 1.33.1," https://buildmedia.readthedocs.org/media/pdf/libuv/stable/libuv.pdf, 2019, [Online; accessed 18-November-2019].

[10] O. Foundation, "Don't Block the Event Loop (or the Worker Pool)," https://nodejs.org/uk/docs/guides/dont-block-the-event-loop/, [Online; accessed 13-December-2019].

[11] ProgrammerSought, "Libuv thread pool and thread communication source code analysis," https://www.programmersought.com/article/88851058916/, 2018, [Online; accessed 16-December-2020].

[12] "Thread pool work scheduling," https://github.com/nodejs/node/blob/ea736668500d640783242186334c38a144501961/deps/uv/docs/src/threadpool.rst, 2019, [Online; accessed 16-December-2020].

[13] "Node.js v14.4.0 Documentation - Modules," https://nodejs.org/api/modules.html, [Online; accessed 15-June-2020].

[14] D. Suleiman, M. Ibrahim, and I. Hamarash, "Dynamic voltage frequency scaling (dvfs) for microprocessors power and energy reduction," in *4th International Conference on Electrical and Electronics Engineering*, vol. 12, 2005.

[15] R. J. Wysocki, "CPU Performance Scaling," https://www.kernel.org/doc/html/v4.12/admin-guide/pm/cpufreq.html, 2017, [Online; accessed 8-December-2020].

[16] G. Dhiman and T. S. Rosing, "Dynamic voltage frequency scaling for multi-tasking systems using online learning," in *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED'07)*. IEEE, 2007, pp. 207–212.

[17] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Proceedings of the 2004 international symposium on Low power electronics and design*, 2004, pp. 174–179.

[18] B. Rountree, D. K. Lowenthal, M. Schulz, and B. R. De Supinski, "Practical performance prediction under dynamic voltage frequency scaling," in *2011 International Green Computing Conference and Workshops*. IEEE, 2011, pp. 1–8.

[19] J. Choi, B. Jung, Y. Choi, and S. Son, "An adaptive and integrated low-power framework for multicore mobile computing," *Mobile Information Systems*, vol. 2017, 2017.

[20] B. Acun, K. Chandrasekar, and L. V. Kale, "Fine-grained energy efficiency using per-core dvfs with an adaptive runtime system," in *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 2019, pp. 1–8.

[21] D. Li and W. G. Halfond, "Optimizing energy of http requests in android applications," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, 2015, pp. 25–28.

[22] A. B. Lago and I. Larizgoitia, "An application-aware approach to efficient power management in mobile devices," in *Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, 2009, pp. 1–10.

[23] W. Cui, D. Richins, Y. Zhu, and V. J. Reddi, "Tail latency in node. js: energy efficient turbo boosting for long latency requests in event-driven web services," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2019, pp. 152–164.

[24] D. Beyer and P. Wendler, "Cpu energy meter: A tool for energy-aware algorithms engineering," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 126–133.

[25] X. Liu, P. Shenoy, and M. D. Corner, "Chameleon: Application-level power management," *IEEE Transactions on Mobile Computing*, vol. 7, no. 8, pp. 995–1010, 2008.

[26] I. Corporation, "Intel 64 and IA-32 Architectures Software Developers Manual Volume 3B: System Programming Guide, Part 2," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf, 2021, [Online; accessed 29-January-2021].

[27] chih's blog, "Energy measurements in Linux," https://blog.chih.me/read-cpu-power-with-RAPL.html, 2017, [Online; accessed 29-January-2021].

[28] M. P. Dimitrov, "Intel Power Governor," https://software.intel.com/content/www/us/en/develop/articles/intel-power-governor.html, 2012, [Online; accessed 29-January-2021].

[29] O. Foundation, "Express," https://expressjs.com/, 2017, [Online; accessed 1-February-2021].

[30] M. Fowler, *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012.

[31] M. Eernisse, "EJS - Embedded JavaScript templates," https://ejs.co/, 2021, [Online; accessed 1-February-2021].

[32] I. MongoDB, "MongoDB," https://www.mongodb.com/, 2021, [Online; accessed 12-February-2021].

[33] ——, "MONGODB MANUAL — Introduction to MongoDB," https://docs.mongodb.com/manual/introduction/, 2021, [Online; accessed 12-February-2021].

[34] O. Collective, "Mongoose," https://mongoosejs.com/, 2021, [Online; accessed 1-February-2021].

[35] "dbkoda-data - Sample collections for MongoDB," https://github.com/SouthbankSoftware/dbkoda-data, 2021, [Online; accessed 29-January-2021].

[36] H. O. Data, "Unemployment Data," https://opendata.hawaii.gov/dataset/unemployment-data, 2020, [Online; accessed 1-February-2021].

[37] ——, “Statewide Unemployment,” https://opendata.hawaii.gov/dataset/statewide-unemployment, 2020, [Online; accessed 1-February-2021].
[38] “Node.js v15.7.0 Documentation - Crypto),” https://nodejs.org/api/crypto.html, 2021, [Online; accessed 1-February-2021].
[39] K. Miettinen, *Nonlinear multiobjective optimization*. Kluwer Academic Publishers, 1999.
[40] I. Corporation, “Intel Core i7-8700 Processor,” https://ark.intel.com/content/www/us/en/ark/products/126686/intel-core-i7-8700-processor-12m-cache-up-to-4-60-ghz.html, 2021, [Online; accessed 3-February-2021].
[41] WikiChip, “Core i7-8700 - Intel,” https://en.wikichip.org/wiki/intel/core_i7/i7-8700#:~:text=The%20i7%2D8700%20operates%20at,burst%20frequency%20of%201.2%20GHz, 2021, [Online; accessed 4-February-2021].
[42] R. Jejurikar and R. Gupta, “Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems,” in *Proceedings of the 2004 international symposium on Low power electronics and design*, 2004, pp. 78–81.
[43] W.-Y. Liang, S.-C. Chen, Y.-L. Chang, and J.-P. Fang, “Memory-aware dynamic voltage and frequency prediction for portable devices,” in *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2008, pp. 229–236.