2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS) | 978-1-6654-5838-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/MASCOTS53633.2021.9614299

Energy-Efficiency Comparison of Common Sorting Algorithms

Norbert Schmitt
Chair of Software Engineering
University of Würzburg
Würzburg, Germany
norbert.schmitt@uni-wuerzburg.de

Supriya Kamthania
Hewlett Packard Enterprise
Bengaluru, India
supriya.kamthania@hpe.com

Nishant Rawtani
Hewlett Packard Enterprise
Bengaluru, India
nishant.rawtani@hpe.com

Luis Mendoza
Hewlett Packard Enterprise
Houston, USA
luis.mendoza@hpe.com

Klaus-Dieter Lange
Hewlett Packard Enterprise
Houston, USA
klaus.lange@hpe.com

Samuel Kounev
Chair of Software Engineering
University of Würzburg
Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

***Abstract*—With the rising demand for information technology comes an increase in energy consumption to power it. Namely, cloud computing is constantly growing, in part due to a rising number of devices using the cloud to provide certain functionality. This growth leads to an increase in energy consumption in data centers and is estimated to climb to over 1PWh in 2030. Hardware manufacturers counter the rising demand for energy in cloud data centers by providing techniques to make servers more energy-efficient. However, the advances cannot fully compensate for the growth. To further increase energy efficiency, software needs to be addressed as well but is often neglected by the developers.**

In this paper, we compare six sorting algorithms, a common task in most programs, against each other in terms of energy efficiency to allow developers to select the best solution to their problem. We selected well-known algorithms in two variants, and two implementation languages, C and Python. We ran each algorithm on two, out of four, state-of-the-art server systems with different CPUs.

***Index Terms*—Energy Efficiency, Sorting Algorithms, Power Consumption, Software Efficiency, Cloud, Data Center, Green Computing, Performance**

I. Introduction

A growing number of services are deployed in the cloud. Additionally, an increasing number of devices are using the cloud to provide functionalities and enabling data centers to grow. This growth, in turn, results in a rise in energy demand with a total energy consumption of 263 TWh annually in 2020 expected to grow to an estimated 1137 TWh by 2030 [1]. Moreover, while advances in hardware increase their energy efficiency for sustainable growth, they cannot fully compensate for the increase in energy demand in data centers [2]. Therefore, it is essential to include software as a significant part of conserving energy and increasing energy efficiency.

Fortunately, cloud data centers can be made more efficient. Many technologies exist to increase the energy efficiency of data centers on the hardware level, like dynamic voltage and frequency scaling (DVFS), and benchmarks to create an incentive for manufacturers to improve energy efficiency, such as the SPECpower_ssj® 2008 [3]. For software, energy can be conserved by intelligently placing or consolidating deployed software [4] or using only the minimal amount of resources to satisfy user demands through auto-scaling [5]. However, the software itself needs to be designed and implemented with energy efficiency in mind.

In a 2016 study by Pang et al. about software energy consumption, out of 122 programmers, only 18% answered that they take energy consumption into account when writing software. More positively, 14% considered minimizing energy consumption a requirement, and 21% responded they already changed the software to consume less energy [6]. This survey shows that optimizing energy consumption is not widely considered and adapted even if it does not mean trading energy consumption for lower performance, as Capra et al. showed by testing two different enterprise resource planning (ERP) systems. While both had similar performance differing by about 5%, their energy efficiency varied by about 50% [7]. Pinto et al. also argue that energy consumption is becoming a more critical aspect of software engineering and development, but there is a lack of knowledge on achieving less energy consumption by software [8].

Closing this knowledge gap is essential. One important aspect is selecting a good or even the best algorithm for a given task. One task that probably is part of most programs is sorting data. Sorting algorithms are well known, and some research exists focusing on conserving energy by looking at those algorithms [9]–[11]. However, they put their focus either on battery-powered mobile devices [9], [10] or on comparing different data types (floating-point versus integer) of a small problem size on desktop computers [11].

Our contribution in this paper is the analysis of the energy efficiency of six common sorting algorithms. We use state-of-the-art server systems, two different programming languages, and two different implementation variants for Python and C to analyze which or if an approach is more energy-consuming

than another. We also compiled a list of guidelines for energy efficient sorting. Industry practitioners in software development then could utilize our results to make informed decisions about the most suitable sorting algorithms for their problem at hand.

The remainder of this paper is structured as follows. We first give an overview of related work in Section II, followed by the description of our testbed setup, selected algorithms, and approach on the measurements in Section III. We then evaluate the energy efficiency of common sorting algorithms in Section IV and present the compiled guidelines in Section V. Afterwards threats to validity are discussed in Section VI and we conclude our work in Section VII.

II. Related Work

Much research has been conducted regarding power consumption, energy consumption, and energy efficiency. In this section, we will give an overview of works related to energy consumption, decreasing energy wastage, and increasing energy efficiency, as well as the energy efficiency of algorithms. This section is based partially on previous work [12].

For mobile devices power and energy consumption are pressing issues, primarily if the devices are powered by batteries. In [13] Li et al. present an approach to measure power consumption on a source line level. The authors combine hardware-based power measurements with program analysis and statistical modeling. While executing an application, the approach measures the energy and derives the executed parts of the application using path profiling. The per-line consumption is then evaluated using static and regression analysis as presented in the authors' previous works [14]. However, this approach requires a significant amount of resources to perform the profiling, reducing accuracy.

Pathak et al. introduce *eprof*, a fine-grained energy profiler for smartphone apps [15]. When applied to several apps, eprof exposes energy drainers like third-party advertisements or pinpoints wake lock bugs in the code. Next, *bundles* are introduced to help developers optimize their app's energy drain by presenting the app's I/O energy consumption.

Another tool-assisted approach by Zhang et al. [16] introduces two tools. *PowerBooter* constructs a power model without using a power meter. *PowerTutor* is a tool that, using online analysis, shows developers the implications of their design choices on power consumption. Unlike [13], this approach is not based on the code line level but instead on the component level. Thus, the model is built using the consumption of the CPU, LCD, GPS, Wi-Fi, 3G, and audio components.

For Java-based software systems, Seo et al. [17] present the first iteration of a framework for estimating power consumption. This approach focuses on the interaction among distributed components. Thus, it allows developers to estimate their systems' power consumption at design time. However, the component-based approach is coarse, making informed decisions for developers of single components complicated due to the high abstraction.

Banerjee et al. [18] generate an event flow graph of a mobile application and trace the execution to find pieces of code related to high energy consumption. They distinguish between energy hotspots, shorter peaks, and energy bugs that result in increased energy consumption over extended periods of time. This approach aids the developers in showing them that energy hotspots and bugs are present and locates them in the code.

These works on mobile devices have in common that they do not directly use CPU performance counters but use either the sensors of a mobile device, tracing, profiling, or Java's bytecode. While this might work for the specific device or language and gives insight into where energy is wasted, it is complicated to derive accurate, practical guidance without intensive measurement setups [15], [18] for developers to decrease energy wastage and increase energy efficiency.

Other works shift their focus away from modeling the power and energy consumption of devices and relate to the energy-efficiency of software without regard to what type of hardware is executing the application. As a good example, Capra et al. measured and analyzed complete software systems: two ERP systems, two customer relationship management (CRM) systems, and four database management systems (DBMS). The authors could show that software can differ considerably (50%) in energy efficiency without significant differences in performance (5%) in extreme cases. Further, they found that this discrepancy stems from using hibernation during external requests and that algorithmically more efficient software also can be more energy-efficient [7].

Consequently, Aggarwal et al. [19] developed *GreenAdvisor* based on their previous work [20] about system-calls as their primary source to predict power consumption. GreenAdvisor is a tool that analyzes the appearance of system calls in an application and maps them to the corresponding function call inside the application. The change of system call behavior across multiple versions is tracked to determine if a significant change in energy consumption has occurred.

Stier et al. modeled the power consumption of software systems with the Palladio Component Model on an architectural level, achieving an error of less than 5.5% [21]. Nevertheless, modeling the power consumption, an abstraction itself, on a high level can lead to a better architecture but is too coarse to make informed decisions for developers of single components of a software system, similar to Seo et al. [17]. Hence, it can achieve only an improvement on the architecture level but not on an algorithmic level. For this reason, we selected sorting algorithms for our analysis as we argue that sorting is a problem common among many applications.

Bunse et al., as in our work, used measurements of six sorting algorithms, with some in both iterative and recursive implementation, to define a trend function, allowing the developers to choose between performance and energy consumption. They continued by using this trend function to increase the battery lifetime of mobile devices [10]. Rashid et al. compared sorting algorithms on an ARM platform, popular for mobile devices, showing that the algorithm and language do affect the energy consumption [9]. While an interesting

and promising approach, in our opinion, mobile devices often are connected to the cloud to fulfill heavy computational tasks, that while conserving energy for a longer battery lifetime, they shift the energy consumption into data centers.

Chandra et al. conducted a basic study on the power consumption, energy consumption, and runtime of sorting algorithms for their work. They found that the energy consumed is related directly to its time complexity and that integer sorting takes less energy than sorting floating-point data [11]. Again, their focus is not on server systems but desktop machines and a small problem size of just 10,000 integer or floating-point numbers.

It can be seen that the focus of energy-efficiency research is on mobile devices, and there is a gap of knowledge on data center hardware. Therefore, we selected state-of-the-art server systems for our measurements. Additionally, most works are concerned with modeling the power or energy consumption and finding the location inside an application where energy is wasted. The actual improvements in energy efficiency often are left to the compiler leveraging compiler performance optimizations. The following related work is taken and adapted from our previous work [22].

Compilers usually are not targeting energy efficiency with the available optimizations; their main target is performance. Hence, to optimize for energy efficiency, the correct optimization settings must be known to adapt a compiler to improve energy efficiency [23], or there must be adaptation points added during compilation. We also took a first look at two different properties, programming language and application domain, of the SPEC CPU 2017 benchmark suite and which property could be responsible for higher energy efficiency when different compiler optimizations are used [24].

Nobre et al. [25] are changing the order in which performance optimizations are performed during compilation. They have shown that while an increase in energy efficiency is possible, the sequence of optimization is dependent on the application and not necessarily transferable to other applications. Kandemir et al. [26] focused on six loop optimizations of a compiler and tested them on five applications and simulations with mixed results. Most loop optimizations increased the power consumption in the embedded systems.

The tool *Socrates* from Gadioli et al. [27] not only tries to find suitable compiler optimizations but also weaves in code that can be used for tuning the software to different targets, such as power consumption or performance. The additional instructions or parameters are used, for example, to adapt the number of OpenMP threads. Hsu et al. [28] inserted instructions to control the Dynamic Voltage Scaling (DVS). The approach selects program parts suitable for running with lower voltage and frequency without degrading performance above a user-selectable value resulting in energy savings of up to 28% and performance degradation of just 5%, increasing energy efficiency.

Our work extends the related work by aiming not at mobile devices but cloud servers. Additionally, our work is intended as a guideline or help for practitioners to select a better choice of sorting algorithm for energy efficiency before compilation and to leave the fine-tuning to approaches that leverage compiler optimizations.

III. Approach

We first describe our used system under test (SUT) and setup, followed by the measurement analysis in which we compare six sorting algorithms in two variants against each other. In a second step we take a look at two different state-of-the-art server systems to analyze if the energy efficiency is transferable or is an attribute of the server. We make all measurements with two different programming languages, C and Python.

A. System Under Test and Setup

As our SUTs, we selected four different state-of-the-art HPE ProLiant servers, each with a different CPU size. All servers are equipped with the same 480GB SSD storage drive. CentOS 8 is used as the operating system. These servers are, in our opinion, a good representation of standard cloud servers that are not purpose built and do not contain any additional acceleration units. Each server's power supply is connected to a Yokogawa WT210 power analyzer for power measurements. The power analyzer samples internally at about 10kHz and aggregates each sample over one second. The setup is in accordance with the power and performance benchmarking methodology described by Kounev et al. [29]. For our analysis, we do not use the internal sampling rate but the aggregated sampling rate of one second.

TABLE I
SYSTEMS UNDER TEST.

Name	Cores/Threads	Clock	Memory
Server A	128/256	2.25GHz	16x16GB
Server B	56/112	2.20GHz	12x16GB
Server C	32/64	3.00GHz	16x16GB
Server D	36/72	2.60GHz	12x16GB

B. Selected Sorting Algorithms

We have selected six commonly known sorting algorithms, listed in Table II. The algorithms were chosen as they have different runtime behaviors, with a range of n^2, n, and $n \log(n)$ for their best, average, and worst case respectively. While Merge and Heap Sort have identical time complexity of $n \log(n)$, they differ in space complexity. Stability of an algorithm has not been considered. We also did not consider hybrid sorting algorithms like Tim Sort or Intro Sort that combine techniques from algorithms already in our list.

The authors self-implemented each algorithm in two variants with their differences shown in Table III to avoid bias of a specific implementation variant, and two programming languages, Python and C. We selected Python and C as they are well known representatives for interpreted and compiled

languages[1,2]. Yet it must be kept in mind that the results might not be transferable to other languages without further measurements.

TABLE II
SELECTED COMMON SORTING ALGORITHMS.

Name	Time Complexity			Space
	Best	Average	Worst	Complexity
Merge Sort	$n \log(n)$	$n \log(n)$	$n \log(n)$	n
Heap Sort	$n \log(n)$	$n \log(n)$	$n \log(n)$	1
Quick Sort	$n \log(n)$	$n \log(n)$	n^2	$\log(n)$
Insertion Sort	$n \log(n)$	n^2	n^2	1
Bubble Sort	n	n^2	n^2	1
Selection Sort	n^2	n^2	n^2	1

To stress the SUT, each algorithm had to sort a number of integers, which we refer to as the problem size. For each algorithm, we selected three problem sizes shown in Table IV. The problem size is split across the available threads of the SUT listed in Table I. The problem sizes shown in these tables for each algorithm were calibrated separately based on the average time complexity characteristics. Setting common problems sizes across algorithms, servers, and implementation variants proved unfeasible because setting small problem sizes for algorithms with linear time complexity, like Heap Sort, reduces run times and not giving sufficient time for data collection from the power analyzer. Similarly, setting large problem sizes for algorithms with quadratic time complexity, like Bubble Sort, increases the runtime to impractical lengths. The problem sizes used for C and Python are different as we would expect a performance contrast between them. To allow for stable measurements of the C algorithms that are roughly equal in length than the Python measurements, the problem size would have to be larger than the memory and would distort the measurement due to disc IO. We, therefore, took actions, such as reducing the problem size, to achieve a stable measurement while at the same time keeping disc IO to a minimum.

The total problem sizes used for the SUTs are kept constant. For example, if we calibrated the Python implementations of Insertion Sort to a total of 10,240,000 (see Table IV), all integers that must be sorted are distributed across the cores equally. If we want to sort 10,240,000 integers on server A with 256 threads and server B with 112 threads, each thread of server A is given 40,000 (10,240,000/256) integers and each thread of server B is given 91,428 (10,240,000/112) integers. Pseudo-random numbers were generated by reading from the `urandom` file for the C implementations and using the `random` library for the Python implementations. All problem sizes fit in the memory of the SUTs.

C. Analysis

Measurements are taken on four SUTs. For each sorting algorithm, two variants in two different programming languages, and three problem sizes are used and run on two SUTs, resulting in a total of 144 measurements. For Python, the servers A and B have been used, and for C, the servers C and D. Each experiment has been measured five times. The aim for this paper is to compare the energy efficiency against each other to determine which implementation approach is more preferable under certain circumstances, different servers, implementation variant, and problem sizes.

The energy consumption is calculated by integrating over the measured power consumption. For our sampling interval of one second, this is the sum of the power consumption P in Equation 1. To calculate energy efficiency shown in Equation 2, the energy consumption is calculated by dividing the problem size p (total number of sorted integers) by the total energy consumption E, resulting in the number of integers sorted by Joule. For the number of bytes sorted by Joule, we multiply p by four, the size of an integer on the SUTs. We count a variant as preferable if the 95% confidence intervals do not overlap.

$$E = \sum P \tag{1}$$

$$\mathit{Eff} = \frac{p * 4}{E} \tag{2}$$

IV. ANALYSIS

First, we will take a look at the algorithms that do not have a logarithmic runtime behavior. It is clear from the results in Table V and VI that the three sorting algorithms that have an average time complexity of n^2, Bubble Sort, Insertion Sort, and Selection Sort, are much less energy efficient than the algorithms with $n * \log(n)$ as expected no matter the implementation variant.

Regarding the three algorithms with $n * \log(n)$ time complexity, Heap Sort, Merge Sort, and Quick Sort, the Heap Sort algorithm is outperformed across almost all implementation variants, problem sizes, and SUTs (see Table V). The only exception is Quick Sort on server A, variant 2 on a small problem size with 62MB per Joule compared to 64MB per Joule for Heap Sort. We can also determine that even an inefficient implementation to generate the max heap for Heap Sort is not the main reason for this behavior as variant 1 and 2 achieve similar energy efficiency values on all problem sizes and both SUTs. Merge Sort maintains a stable energy efficiency across the problem sizes for each server. One exception is variant 2 on server A with a medium problem size close to 62MB per Joule but still in the same range of Quick Sort and outperforming Heap Sort given this configuration. For Heap Sort on server A with a higher thread count, inlined code (variant 1) has higher efficiency. At the same time, not inlining and calling additional functions seems beneficial for servers with smaller core counts. Quick Sort scales well on server A and becomes more energy-efficient the larger the problem size grows. Although, on server B, it seems to plateau as the configurations for the problem sizes are similar by the 95% confidence interval. In general, Quick Sort also has a higher energy efficiency than Merge Sort for server B.

[1]https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020. Accessed: May 3rd, 2021

[2]https://www.tiobe.com/tiobe-index/. Accessed: June 18th, 2021

TABLE III
IMPLEMENTATION VARIANTS FOR PYTHON.

	Python		C	
Algorithm	Variant 1	Variant 2	Variant 1	Variant 2
Merge Sort	Recursive and code inlined	Recursive and code distributed across multiple functions	Recursive and without dynamic memory allocation	Recursive and with dynamic memory allocation
Heap Sort	Efficient iterative loop to build max heap	Inefficient iterative loop to build max heap	Memory based swapping	Pointer based swapping
Quick Sort	Iterative	Recursive	Memory based swapping	Pointer based swapping
Insertion Sort	Iterative	Recursive	Memory based swapping and no separate function for sorting	Memory based swapping but additional function to perform sort operation
Bubble Sort	Iterative	Recursive	Memory based swapping	Pointer based swapping
Selection Sort	Explicitly type checking in return function to ensure the consistency between the element types going in and out of the sorting function	Dynamic return type (no explicit type checking)	Memory based swapping	Pointer based swapping

TABLE IV
CALIBRATED PROBLEM SIZES FOR PYTHON.

		Problem Size (# of integers)		
	Algorithm	Small	Medium	Large
Python	Merge Sort	1,024,000,000	1,152,000,000	1,280,000,000
	Heap Sort	1,280,000,000	1,408,000,000	1,536,000,000
	Quick Sort	1,536,000,000	1,664,000,000	1,792,000,000
	Insertion Sort	10,240,000	11,520,000	12,800,000
	Bubble Sort	10,240,000	11,520,000	12,800,000
	Selection Sort	10,240,000	11,520,000	12,800,000
C	Merge Sort	5,760,000,000	6,120,000,000	6,480,000,000
	Heap Sort	5,760,000,000	6,120,000,000	6,480,000,000
	Quick Sort	5,760,000,000	6,120,000,000	6,480,000,000
	Insertion Sort	3,600,000	7,200,000	10,800,000
	Bubble Sort	3,600,000	7,200,000	10,800,000
	Selection Sort	3,600,000	7,200,000	10,800,000

Essentially, better energy efficiency for a larger thread count can be observed between both SUTs for Python. These results are expected. We assume that this stems from the measurement setup, measuring wall power at the power supply for the complete server. As server A has a higher thread count but, apart from memory, similar hardware, the base power consumption is similar, resulting in a lower energy efficiency score. This lower score is neither attributable to the CPU or the algorithm.

For the C implementations, shown in Table VI, we take a look at only the three algorithms with a time complexity of $n * \log(n)$. It can be observed that Merge Sort outperforms all other sorting algorithms by a large margin if the memory is not dynamically allocated in variant 1. Nevertheless, if the memory is dynamically allocated with variant 2, its energy efficiency is reduced by a factor ranging from 7.3 on the medium problem size and server C up to 13.4 on the large problem size and server D.

Quick Sort is the next most energy-efficient implementation in C and should be preferred over a Merge Sort with dynamically allocated memory. Both variants of Quick Sort are more energy efficient than variant 2 of Merge Sort. Yet, in all but one case (medium problem size on server C), Quick Sort becomes less energy efficient when it is implemented with pointer based swapping. This single case for Quick Sort is probably due to the large confidence intervals and we assume that additional measurement runs would resolve this outlier.

Regarding Heap Sort, the difference between pointer and memory based swapping is not as clear. There are two measurements on server C, the small and large problem size, that do overlap on the 95% confidence interval. Given that Quick Sort has a better space complexity of $\log(n)$ compared to Heap Sort with n, it is expected that Heap Sort is less susceptible to changes on how the swapping in memory is handled.

V. GUIDELINE

We will take the findings from our measurements in Section IV before and provide some basic guidelines for practitioners for each sorting algorithm. We left out Bubble Sort, Insertion Sort, and Selection Sort on purpose as a much higher energy efficiency can be achieved and we would recommend, given the choice, implementing a different algorithm with a better time complexity. We also found that there are major differences even for algorithms of similar time complexity. We marked in front of each guideline whether it is applicable to both implementation languages (*Both*), only for *Python*, or only for *C*.

Merge Sort

Both Merge Sort is preferable over a Heap or Quick Sort algorithm when memory is no constraint.

Python Use Merge Sort for up to 1.2 billion integers (medium problem size).

Python In case of a very large amount of sorting data, a Quick Sort algorithm might be a better choice.

TABLE V
ENERGY EFFICIENCY FOR BOTH PYTHON IMPLEMENTATION VARIANTS IN SORTED KB PER JOULE.

	Server	Problem Size	Variant 1 Mean	Variant 1 95% CI	Variant 2 Mean	Variant 2 95% CI	V1 and V2 Overlap
Merge Sort	A	Small	95 827.81	2219.27	89 306.57	320.12	✗
		Medium	93 869.7	702.76	61 848.02	1050.81	✗
		Large	93 251.27	531.87	86 783.91	305.66	✗
	B	Small	42 599.25	255.33	45 074.79	177.47	✗
		Medium	42 401.23	147.29	44 907.01	178.77	✗
		Large	42 149.87	162.97	45 016.84	200.9	✗
Heap Sort	A	Small	64 238.06	1888.36	64 072.1	2055.87	✓
		Medium	49 276.52	3061.87	49 179.08	3273.6	✓
		Large	50 795.02	3537.18	49 664.16	3817.94	✓
	B	Small	31 546.17	70.99	31 561.86	46.96	✓
		Medium	31 514.58	77.78	31 447.73	86.82	✓
		Large	31 350.52	59.84	31 331.65	52.29	✓
Quick Sort	A	Small	85 200.41	500.45	61 514.99	833.26	✗
		Medium	87 686.18	395.88	62 400.97	1038.0	✗
		Large	119 615.0	670.43	77 710.85	255.19	✗
	B	Small	72 838.23	514.72	49 790.54	361.29	✗
		Medium	72 994.33	715.83	49 872.73	338.38	✗
		Large	73 519.13	707.49	49 995.9	310.38	✗
Insertion Sort	A	Small	585.35	2.82	42.03	0.12	✗
		Medium	400.73	3.4	35.41	1.6	✗
		Large	447.42	3.99	31.76	0.06	✗
	B	Small	102.05	0.32	2120.12	75.03	✗
		Medium	90.15	0.17	7.64	0.92	✗
		Large	81.16	0.06	6.63	0.95	✗
Bubble Sort	A	Small	263.49	17.22	274.77	10.95	✓
		Medium	186.99	22.5	184.75	21.4	✓
		Large	173.83	15.49	175.83	15.11	✓
	B	Small	67.61	58.0	192.27	0.64	✗
		Medium	41.34	0.1	42.13	0.17	✗
		Large	37.09	0.08	32.51	14.55	✓
Selection Sort	A	Small	414.76	48.77	429.13	45.98	✓
		Medium	394.48	2.23	390.1	4.08	✓
		Large	410.4	17.12	414.7	11.83	✓
	B	Small	100.39	0.26	102.08	0.36	✗
		Medium	88.82	0.36	90.46	0.22	✗
		Large	79.44	0.38	81.35	0.12	✗

C Preallocate memory if possible. If the memory must be allocated dynamically, a Quick Sort implementation should be preferred.

Heap Sort

Both Implement a Quick Sort algorithm instead when possible and the space complexity is no hard constraint.

Python Max heap generation can be neglected in terms of energy efficiency.

Python Does not scale as good with the growing size of the sorting problem.

Quick Sort

Python Implement iterative variant if possible to achieve a better energy efficiency.

Python It scales better with the growing size of the sorting problem.

C Pre-allocate memory if possible.

VI. THREATS TO VALIDITY

The threats to validity are discussed in the order of severity rated by the authors.

a) Lower number of repetitions: All measurements were repeated five times. However, a higher number of repetitions might reduce the variance and, therefore, some confidence intervals might not overlap, which will negate the conclusions drawn from the measurements. As this work is intended as a guideline, a selection of a good algorithm instead of the optimal still can be of value.

b) Selection of sorting algorithms: The selection of algorithms to choose for this work has been based on the time and space complexity of the algorithms. We are aware that alternatives with similar time and space complexity might exist and argue that, given the assumption that they were implemented with care and perform as intended, they should be interchangeable. Stability, on the other hand, was not

TABLE VI
ENERGY EFFICIENCY FOR BOTH C IMPLEMENTATION VARIANTS IN SORTED KB PER JOULE.

	Server	Problem Size	Variant 1 Mean	Variant 1 95% CI	Variant 2 Mean	Variant 2 95% CI	V1 and V2 Overlap
Merge Sort	C	Small	3 057 140.43	68 234.82	401 219.46	1867.08	✗
		Medium	2 874 856.35	607 613.69	393 527.45	20 993.1	✗
		Large	3 406 575.21	109 261.19	376 680.4	41 104.92	✗
	D	Small	3 554 848.68	702 507.71	322 805.22	1190.23	✗
		Medium	3 946 797.65	572 690.29	318 950.61	3342.17	✗
		Large	4 263 224.3	261 687.77	317 969.18	3109.83	✗
Heap Sort	C	Small	156 507.34	9793.82	156 538.77	2647.3	✓
		Medium	159 005.12	491.7	156 300.54	1164.83	✗
		Large	156 701.09	2314.56	153 358.12	4113.29	✓
	D	Small	112 808.83	165.06	107 845.36	110.66	✗
		Medium	112 207.69	350.81	107 072.05	135.36	✗
		Large	111 542.68	403.48	106 421.56	474.03	✗
Quick Sort	C	Small	712 085.62	2818.19	581 176.35	49 584.84	✗
		Medium	687 025.63	80 615.43	583 890.48	44 280.08	✓
		Large	689 021.31	74 343.39	604 550.75	6370.04	✗
	D	Small	539 591.67	6980.36	424 654.5	5686.78	✗
		Medium	544 565.18	7796.15	420 779.36	3926.93	✗
		Large	545 010.71	5673.11	424 777.04	3415.2	✗
Insertion Sort	C	Small	1571.49	23.95	1299.39	13.67	✗
		Medium	1136.63	11.74	851.6	5.85	✗
		Large	819.89	5.01	600.31	2.2	✗
	D	Small	1565.2	108.45	1328.51	29.46	✗
		Medium	1017.02	24.45	829.03	11.5	✗
		Large	711.04	11.77	578.79	6.98	✗
Bubble Sort	C	Small	464.87	2.79	447.34	2.23	✗
		Medium	249.44	1.82	237.7	0.37	✗
		Large	165.24	1.01	157.21	0.72	✗
	D	Small	368.83	2.2	349.39	2.87	✗
		Medium	192.95	0.88	181.93	1.16	✗
		Large	127.86	0.52	121.8	1.31	✗
Selection Sort	C	Small	571.07	3.53	901.45	5.29	✗
		Medium	314.75	1.36	537.04	3.14	✗
		Large	211.29	1.31	369.8	0.79	✗
	D	Small	497.21	2.6	837.31	23.62	✗
		Medium	264.64	1.39	477.16	4.57	✗
		Large	176.83	1.09	320.51	1.09	✗

considered and the results might not be generalizable to algorithms with different stability properties.

c) Programming language selection: We selected the programming languages Python and C as representatives for an interpreted and compiled language, respectively. Both languages are publicly available for extended periods of time, well-known, and widely used. We are aware that the compilation can introduce optimizations but argue that this is not a threat as compiler optimizations, ahead-of-time or just-in-time, also might be used in a real-world scenario.

d) Problem size selection: We took care to select three different problem sizes for each algorithm that allowed for a stable measurement while keeping the measurement feasible timewise. However, due to the nature of the time complexity of slow algorithms, not all algorithms have been calibrated to identical problem sizes. This threat might invalidate our conclusions when absolute energy consumption results are compared but not for the normalized energy efficiency.

e) Limited number of server configurations: As measuring takes time and the number of possible server configurations is vast, considering all configurations becomes practically infeasible. We, therefore, opted for four configurations of state-of-the-art servers as representative of a cloud data center scenario. As the four options consider both major x86 CPU manufacturers, we see this as a good starting point for this analysis, although, it is possible that servers with a lower computational power might yield different results. Other instruction set architectures have not been considered as, in our opinion, cloud hardware predominantly uses x86 servers, so our generalizability is limited to x86 only.

VII. CONCLUSION

Data centers consume large amounts of energy today, and their energy demand will grow in the future. Energy efficiency needs to be improved to reduce energy consumption in data

centers. While many advances are made for the hardware used, they cannot compensate for the data center growth. The software, on the other hand, often is not considered by developers. There is also some lack of knowledge on how to improve the energy efficiency of software. One aspect is to select a suitable algorithm for the problem. Yet most work on energy-efficient development is concerned with battery-powered mobile devices. Automated approaches using compiler optimizations can be used to improve energy efficiency, but an inefficient algorithm can be optimized only to a certain degree. We therefore analyzed six common sorting algorithms for energy consumption to compile lessons learned.

For our analysis, we selected common sorting algorithms and implemented two variants for each algorithm. Each implementation variant is then measured on two state-of-the-art servers five times with three different problem sizes. Our results show that, even with identical average time complexity of $n * \log(n)$, there are differences between algorithms in how energy efficient they are. We compiled the first version of guidelines for developers and plan to extend and generalize these guidelines in the future, including additional programming languages, SUTs, and algorithms. Additional future work is to confirm our experiments on a real-world application.

References

[1] A. Andrae and T. Edler, "On Global Electricity Usage of Communication Technology: Trends to 2030," *Challenges*, vol. 6, no. 1, pp. 117–157, Apr 2015.

[2] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, "Recalibrating global data center energy-use estimates," *Science*, vol. 367, no. 6481, pp. 984–986, 2020.

[3] SPEC, "Beyond performance to power consumption," https://www.spec.org/30th/power.html, Accessed: 06.10.2020, 2019.

[4] Y. Jin, Y. Wen, and Q. Chen, "Energy Efficiency and Server Virtualization in Data Centers: An Empirical Investigation," in *2012 IEEE Conference on Computer Communications Workshops*, Mar. 2012, pp. 133–138.

[5] N. Herbst, "Methods and benchmarks for auto-scaling mechanisms in elastic cloud environments," Ph.D. dissertation, University of Würzburg, Germany, July 2018.

[6] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016.

[7] E. Capra, C. Francalanci, and S. A. Slaughter, "Is software "green"? application development environments and energy efficiency in open source applications," *Information and Software Technology*, vol. 54, no. 1, pp. 60 – 71, 2012.

[8] G. Pinto and F. Castor, "Energy efficiency: A new concern for application software developers," *Commun. ACM*, vol. 60, no. 12, p. 68–75, Nov. 2017. [Online]. Available: https://doi.org/10.1145/3154384

[9] M. Rashid, L. Ardito, and M. Torchiano, "Energy consumption analysis of algorithms implementations," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015, pp. 1–4.

[10] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, "Choosing the" best" sorting algorithm for optimal energy consumption." in *ICSOFT (2)*, 2009, pp. 199–206.

[11] T. B. Chandra, V. Patle, and S. Kumar, "New horizon of energy efficiency in sorting algorithms: green computing," in *Proceedings of National Conference on Recent Trends in Green Computing. School of Studies in Computer in Computer Science & IT, Pt. Ravishankar Shukla University, Raipur, India*, 2013, pp. 24–26.

[12] N. Schmitt, L. Iffländer, A. Bauer, and S. Kounev, "Online power consumption estimation for functions in cloud applications," in *2019 IEEE International Conference on Autonomic Computing (ICAC)*, 2019, pp. 63–72.

[13] D. Li, S. Hao, W. G. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 78–89.

[14] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 92–101.

[15] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 29–42.

[16] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2010, pp. 105–114.

[17] C. Seo, S. Malek, and N. Medvidovic, "An energy consumption framework for distributed java-based systems," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 421–424.

[18] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 588–598. [Online]. Available: https://doi.org/10.1145/2635868.2635871

[19] K. Aggarwal, A. Hindle, and E. Stroulia, "Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 311–320.

[20] K. Aggarwal, C. Zhang, J. C. Campbell, A. Hindle, and E. Stroulia, "The power of system call traces: predicting the software energy consumption impact of changes." in *CASCON*, vol. 14, 2014, pp. 219–233.

[21] C. Stier, A. Koziolek, H. Groenda, and R. Reussner, "Model-based energy efficiency analysis of software architectures," in *Software Architecture*, D. Weyns, R. Mirandola, and I. Crnkovic, Eds. Cham: Springer International Publishing, 2015, pp. 221–238.

[22] N. Schmitt, J. Bucek, J. Beckett, A. Cragin, K.-D. Lange, and S. Kounev, "Performance, power, and energy-efficiency impact analysis of compiler optimizations on the spec cpu 2017 benchmark suite," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 292–301.

[23] J. Pallister, S. J. Hollis, and J. Bennett, "Identifying compiler options to minimize energy consumption for embedded platforms," *The Computer Journal*, vol. 58, no. 1, pp. 95–109, 2013.

[24] N. Schmitt, J. Bucek, K.-D. Lange, and S. Kounev, "Energy eficiency analysis of compiler optimizations on the spec cpu 2017 benchmark suite," in *Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering (ICPE 2020)*, 2020.

[25] R. Nobre, L. Reis, and J. M. P. Cardoso, "Compiler phase ordering as an orthogonal approach for reducing energy consumption," *CoRR*, vol. abs/1807.00638, 2018.

[26] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of compiler optimizations on system power," in *Proceedings of the 37th Annual Design Automation Conference*, ser. DAC '00. New York, NY, USA: ACM, 2000, pp. 304–307.

[27] D. Gadioli, R. Nobre, P. Pinto, E. Vitali, A. H. Ashouri, G. Palermo, J. Cardoso, and C. Silvano, "Socrates - a seamless online compiler and system runtime autotuning framework for energy-aware applications," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1143–1146.

[28] C.-H. Hsu and U. Kremer, "The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 38–48.

[29] S. Kounev, K.-D. Lange, and J. von Kistowski, *Systems Benchmarking*, 1st ed. Springer International Publishing, 2020.