Energy-Efficient Data Processing at Sweet Spot Frequencies

Sebastian Götz[1], Thomas Ilsche[1], Jorge Cardoso[2], Josef Spillner[1], Uwe Aßmann[1], Wolfgang Nagel[1], and Alexander Schill[1]

[1] Technische Universität Dresden, Germany
[2] University of Coimbra, Portugal
{sebastian.goetz1|thomas.ilsche|josef.spillner|uwe.assmann|wolfgang.nagel|alexander.schill}@tu-dresden.de, jcardoso@dei.uc.pt

Abstract. The processing of Big Data often includes sorting as a basic operator. Indeed, it has been shown that many software applications spend up to 25% of their time sorting data. Moreover, for compute-bound applications, the most energy-efficient executions have shown to use a CPU speed lower than the maximum speed: the CPU *sweet spot* frequency. In this paper, we use these findings to run Big Data intensive applications in a more energy-efficient way. We give empirical evidence that data-intensive analytic tasks are more energy-efficient when CPU(s) operate(s) at sweet spots frequencies. Our approach uses a novel high-precision, fine-grained energy measurement infrastructure to investigate the energy (joules) consumed by different sorting algorithms. Our experiments show that algorithms can have different sweet spot frequencies for the same computational task. To leverage these findings, we describe how a new kind of self-adaptive software applications can be engineered to increase their energy-efficiency.

1 Introduction

Electricity used in global cloud data centers likely accounts for between 1.1% and 1.5% of total electricity use. IDC[3] estimates that the digital universe of Big Data will double every two years, reaching 40,000 exabytes in 2020. This large volume of data will require complex analytics processing applications, which will necessitate massive processing power.

This state of affairs calls for the development of new approaches to reduce energy consumption [1], which has already led researchers to develop new hardware and software architectures, virtual machine migration strategies, and water cooling mechanisms.

Since analytical applications for Big Data rely heavily on the performance of data-intensive tasks such as data aggregation, data sorting, and data formatting,

[3] http://www.emcchannel.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf

in this paper we look into new techniques to make these tasks more energy-efficient. While making, e.g., the task of data sorting 5% more energy-efficient seems a marginal achievement, this value needs to be contextualized (see [2]). Facebook generates more than 5 petabytes of new data on a daily basis; LinkedIn generated more than 5 billion searches in 2012; Twitter generates more than 500 million tweets a day; and Cisco forecast for global mobile data traffic (2013-2018)[4] estimates that traffic will grow at a compound annual growth rate of 78% to 2016, reaching 10.8 exabytes per month. Clearly, reducing in 5% the required energy of a data-intense sorting task translates into large savings.

Our approach involves executing sorting algorithms under various configurations (e.g., the number of elements to sort and the CPU frequency) and evaluating which *sweet spots* of the underlying hardware architecture enables a more energy-efficient execution. The data obtained was used to construct a task profile, which then resulted in a predictive model to determine at which frequency a CPU should be clocked to run a specific data-intensive sorting task most efficiently. We show how to automate the detection of sweet spots and their use to develop self-adaptive software, which automatically selects the CPU frequency to execute algorithms based on sweet spots and energy objectives to reach. Our approach is based on two previous findings:

1. It has been shown that for certain computational applications, the most energy-efficient execution is achieved using the CPU sweet spot frequency, a CPU frequency setting that is often between the minimum and maximum.
2. It has been shown that software applications spend up to 25% of their time sorting data.

1) It is already known from previous academic and industrial research that processors do not follow a proportional path. Single processors have power states and associated frequencies for which the energy-efficiency, i.e., the ratio between utilization and energy consumption is maximized in so-called sweet spots [3] and often minimized in high-performance *turbo mode* [4, 5]. Furthermore, in multi-processor systems, additional overlap effects result from using the turbo mode as a gap-filler before switching on the next core when the utilization increases [6]. This effect, in particular the sweet spot, translates into time-dependent energy efficiency due to Dynamic Voltage Frequency Scaling (DVFS; a commonly used power-management technique).

2) More than 25% of the running time of many applications has at some point been spent on sorting [7]. While more recent results are missing from the literature, it is expected that the processing of Big Data will place an even greater emphases on sorting. In their approach, Herodotou et al. [8] also use sorting algorithms to evaluate the performance of systems for Big Data analytics since "they are simple, yet very representative". Other data-intensive tasks of interest, but not explored in our work, include search and indexing over large volumes of data.

[4] http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html

Among our results, we determined that the automated detection and use of sweet spot frequencies reduces energy consumption by up to 25% and that data-intensive tasks can be designed as self-adaptive applications to exploit the benefits of sweet spots. Our findings can contribute to the development of energy-efficient analytics applications and databases for Big Data, since they operate on large amounts of data and heavily rely on the task of sorting to process data.

This paper is structured as follows. Sect. 2 enumerates three central research questions that are addressed throughout this paper. Sect. 3 describes our approach and the methodology we have followed to experimentally analyze the energy-efficiency of data-intensive processing tasks. We discuss the results of our experiments in Sect. 4 and show in Sect. 5, how these findings can be used, to build self-adaptive software, able to leverage this knowledge to save energy. Finally, Sect. 6 and 7 present related work and our conclusions, respectively.

2 Research Questions

The energy-efficiency of data-intensive processing tasks looks into how software, the underlying computing system and the environment affect energy consumption. Our research questions (RQ) are the following:

- RQ1 (Measurement Setup). How to instrument a computing system with measurement devices to obtain fine-grained measurements for its individual parts (e.g., fan, disk, power supply, and CPU sockets)? (Sect. 3.1).
- RQ2 (Sweet spots). How can sweet spot frequencies contribute to improve the energy-efficiency of data processing tasks? Which mathematical functions characterize them? (Sect. 4).
- RQ3 (Dynamic Software Adaptation). How to explore sweet spots to dynamically adapt software to achieve a higher energy-efficiency? (Sect. 5).

In this paper, we study how different implementations of algorithms typically used by data-intensive tasks affect differently the energy efficiency of a computing system. We take the computational task of sorting n numbers, as it represents a common use case for Big Data analytics.

While research has looked into how to make information and communication technologies more energy-efficient, it is rather hard to find a precise definition for *software energy-efficiency*. Thus, to remove any possible ambiguity on the results of our research, we define the concept as follows:

Definition 1 (Software energy-efficiency) *Energy is defined as the amount of joules, required by a full or partial computing environment, to execute a software application. A software application S_1 is said to be more energy-efficient than an application S_2, if it requires less energy to accomplish the same computational task.*

Definition 2 (Computing environment) *A full computing environment includes all the devices that, directly or indirectly, consume energy to enable a*

software application to be executed. For example, it typically includes CPUs, fans, and disks. A partial computing environment only includes a subset of those devices.

3 Approach and Methodology

Computational complexity (i.e., big O notation) is often a first step in assessing the performance of an algorithm. However, in practice, the best big O algorithm may perform worse due to, e.g., physical memory constraints. Quicksort is such an example since is often used even though it does not have the best big O (worst case) performance. It is common practice to optimize implementations for runtime and, in most cases, optimizations will also reduce energy consumption. However, in recent hardware with increasing energy-efficiency features, such as DVFS, the fastest algorithm is often no longer the most energy-efficient one.

Our approach to gain insights is experimental. We use energy as a main optimization goal and vary the algorithm and hardware configuration for comparison. The methodology has the following activities:

- Measurement environment (Sect. 3.1).
  - Instrument server with energy sensors.
  - Determine static power consumption of the server.
  - Setup software infrastructure to conduct the experiments.
- Software under test (Sect. 3.2).
  - Select the data-intensive computational task to be tested experimentally.
  - Select different software implementations for the task.
- Experimental results analysis (Sect. 4).
  - Determine resources affected by task.
  - Interpret measurement results.
- Generalization and application of the results (Sect. 5).

3.1 Measurement Setup: Energy Monitoring

Hardware The system under test is a dual socket system with Intel Xeon E5-2690 processors. Several layers of power measurement instrumentation are required. The complete AC input is measured with a calibrated ZES Zimmer LMG450 power analyzer. Several custom-built, shunt-based sensors are added to the system. All sensors are pluggable via Molex connectors used in many standardized systems. For this paper, we monitor the 12 V input of the two individual sockets separately. They supply power for the CPUs and their attached memory. The voltage drop over the measurement shunt is amplified with calibrated amplifiers and digitally captured by a National Instruments PCI-6255 data acquisition board with 7541 samples per second. The power consumption is computed digitally from individual readings for current and voltage. During the experiment, all data processing happens on a separate system to avoid perturbation of the system under test. This measurement infrastructure serves as an answer to RQ1, but requires significant effort, as described in [9].

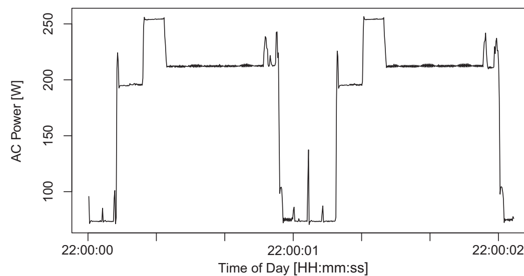


Fig. 1. Energy trace of two succeeding sorting invocations with idle phase.

The processors provide 15 different frequencies from 1.2 to 2.9 GHz and the turbo mode with frequencies up to 3.8 GHz, depending on thermal and power budget. Both, frequency and voltage are set uniformly for all cores of a socket by the hardware. As demonstrated in [10], the available memory bandwidth depends on the core frequency for the Sandy Bridge-EP architecture. Earlier Intel architectures, such as the one used in [3], provided a constant memory bandwidth independent of the selected frequency. The variable memory bandwidth did not allow for a straight-forward selection of the optimal frequency for applications that become memory-bound at a certain frequency. However, our approach does not require specific bottleneck analysis, because it uses the energy measurement results to select among different settings instead.

Measurements To investigate the energy consumption of data-intensive tasks, we run different sort algorithms for different input sizes multiple times. Prior to each invocation, we randomly generate integer lists to be sorted. Across all invocations, we used lists of sizes between 10 and 50 million elements, always containing integers with a value range of 6 million (i.e., $0 \leq x < 6 \times 10^6$). Each invocation is preceded by a pause of 1 second to "cool down" the CPU and other resources (i.e., let them switch to idle mode).

Measurements of sort invocations utilizing a single core of a multi-core machine are not representative, due to the static power consumption of other devices besides the CPU, which does not change, regardless of how many cores are used. Hence, we fully utilized all cores of the machine with a separate sort invocation operating on a copy of the same list. By using MPI barriers [11], we ensure that all sort invocations are started at the same time. As execution time, we measured the longest duration of the parallel sort invocations and ensured that variation of durations among parallel processes was less than 5%.

Fig. 1 visualizes the invocation scheduling by AC power consumption (i.e., at the wall) over time for two consecutive sort invocations. The spikes at the beginning and end of each invocation are due to (MPI) synchronization. The short period of 250 W in each run denotes the list generation.

As shown in Table 1, the static power consumption of the server originates from the power supply, the fan, the motherboard, the disk, and the sockets.

Power supply & Fans	Board	SSD	Sockets	Total
≈26W	≈7W	≈1W	≈20W × 2	≈74W

Table 1. Static idle power consumption of the server.

For the investigation of the effect of different CPU frequencies on the timing and energy-efficiency of sorting, we used the *userspace* CPU governor of Linux to explicitly set the frequency of the CPU. Linux also offers the governors *performance* and *powersave* that statically select the highest/lowest frequency within the borders of a setting. The *ondemand* governor changes the frequency based on CPU usage. In our use case of continuously sorting, the CPU usage will be at 100% for the active cores. Therefore, the *ondemand* governor will eventually select the highest frequency.

We executed list generation and sorting for three algorithms with different list sizes for all possible frequencies of the CPU and the turbo mode. We collected the total energy consumption of the server per execution, the energy consumption of the sockets, and the response time. This enables to investigate whether a sweet spot frequency exists and if static power consumption leads to a shift of the sweet spot frequency for sockets, only compared to the whole server.

3.2 Software Setup: Sorting Algorithm

As mentioned before, data analytics over Big Data are a key target for energy efficiency with huge absolute savings, even for small percentages in relative savings. These systems are very complex and rely heavily on computational tasks such as data cleansing, data aggregation, data sorting, and data formatting. We suggest to study a well-known and representative algorithm: sorting. In many applications, more than 25% of the running time of computers has at some point been spent on sorting [7].

We selected two stable sort implementations: counting sort and radix sort. We also looked at the non-stable sorting of the C++ Standard Library: std::sort. Counting sort is a stable sorting algorithm with a linear time complexity of $O(n)$. Radix sort performs $O(n \times log_2(n))$ comparisons [12]. Our implementation (GNU libstdc++) uses a combination of intro sort and insertion sort.

Based on this knowledge about the time complexity of sorting algorithms, we investigated the energy consumption of the three algorithms. Since sorting is compute bound, one could assume the CPU to be the predominant consumer of energy amongst all other resources in a server. We collected empirical support for this hypothesis and, in addition, determined further resources consuming energy due to sorting. It is important to know that the frequency of a CPU affects timing, power and, consequently, the energy consumption of algorithms. In general, we will give empirical support for the following claims:

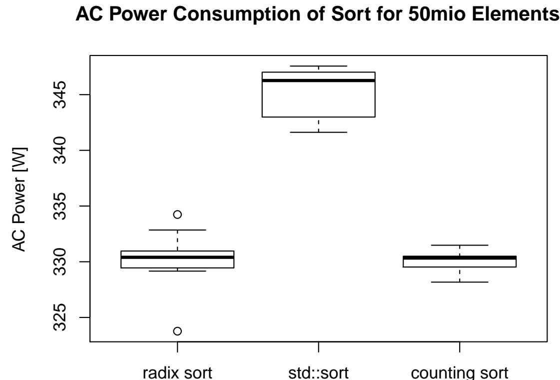


Fig. 2. AC power consumption for sorting 50 million elements.

1. The highest frequency of a CPU does not necessarily lead to the lowest energy consumption (power integrated over time).
2. Each algorithm has a detectable frequency at which the resulting energy consumption is lowest (sweet spot frequency).
3. Different algorithms can have different sweet spot frequencies.

4 Results of the Experiments

Before investigating the impact of different frequencies on energy-efficiency in Sect. 4.2, we analyze in Sect. 4.1[5] whether algorithms differ in their power consumption and show that algorithms with low power consumption are not necessarily the best in terms of energy consumption.

4.1 On the Power Consumption Level of Software

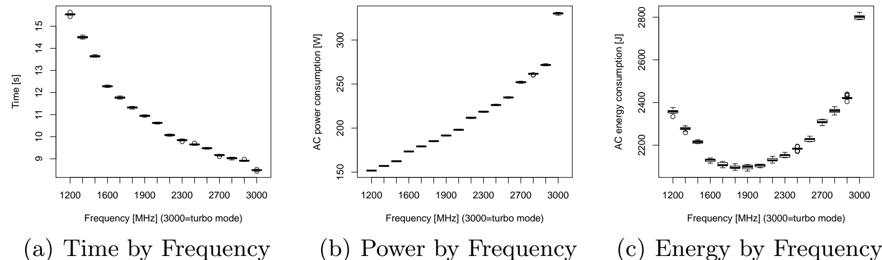
The weighted moments, among them the mean value, quartiles, minimum and maximum excluding extreme outliers, are shown in Fig. 2 for a sorting task on 50 million elements where both CPUs of the server operate in turbo mode.

From the figure, one can infer that using radix or counting sort leads to a lower power consumption than std::sort. The former two have an almost equal level. The savings compared to std::sort amount to 4.6%. Yet, radix sort is clearly the fastest algorithm and, hence, the best when combining both metrics without prioritization of one over the other. Counting sort is, despite its low power consumption, the worst in terms of energy consumption. From this so-called *sweet spot perspective*, which will be further elaborated on in the following section, the savings for radix sort amount to 77.6% compared to counting sort. Table 2 summarizes the key numbers from the experiments: Duration including range,

[5] All raw measurements retrieved from the experiments were processed with the statistics software R.

Table 2. Sorting algorithms comparison for 50mio elements (in turbo mode).

Algo.	t_{min} (s)	t_{max} (s)	t_{range} (s)	P (W)	E (J)
radix	1.880	1.901	0.021	330.4	626.9
std::sort	4.226	4.230	0.004	346.3	1464.1
count.	8.444	8.516	0.072	330.3	2801.5



(a) Time by Frequency (b) Power by Frequency (c) Energy by Frequency

Fig. 3. The sweet spot of counting sort to sort 50mio elements.

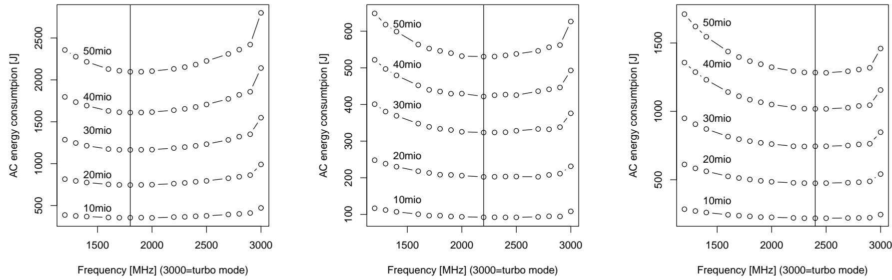
power consumption per time unit and overall energy consumption for the mean duration. All power values include 74 W idle consumption.

4.2 On the Sweet Spots of Software

In this section, we investigate and prove wrong the common misconception that software energy-efficiency equates directly to CPU performance. For this purpose, we have collected evidence that executing software applications at high CPU frequencies may lead to lower software energy-efficiency. Therefore, we refine research question RQ2 into:

- RQ2a (Effectiveness). Can we increase software energy efficiency by changing the clock frequency of the CPU executing a software task?
- RQ2b (Determinability). Can the clock frequency of the CPU executing a software task, which makes the software more energy efficient, be determined?

For the three different algorithmic implementation, radix sort, std::sort and counting sort, we measured the energy consumed to sort 10, 20, ..., 50 million integers. Fig. 3 shows the results obtained for sorting 50 million elements using counting sort. Measurement results for all other list sizes and algorithms are shown in Fig. 4. The figure shows three charts: time per frequency, power per frequency, and energy per frequency. Note that turbo mode frequency varies over time, depends on different factors and can be between 2.9 and 3.8 GHz. Very high frequencies are unlikely as we fully use all cores.



(a) Counting sort (sweet spot at 1.8 GHz) (b) Radix sort (sweet spot at 2.2 GHz) (c) std::sort (sweet spot at 2.4 GHz)

Fig. 4. AC energy consumption and the corresponding sweet spot frequencies.

Time×Freq Fig. 4(a) shows that as the clock frequency of the CPU increases from 1.2 GHz to 2.9 GHz and turbo mode, the mean time required to execute the software task of sorting decreases in a non-linear form. What should be noticed is that at 1.4 GHz the mean time to complete the task drops significantly.

Power×Freq Fig. 4(b) shows the power consumed based on the CPU clock frequency selected. The results could be considered foreseeable, since the power consumption of the CPU increases by its frequency. Nonetheless, up to 2 GHz the power consumption is more modest and has visible increments at 1.5 GHz, 2.2 GHz, 2.7 GHz and for the turbo mode.

Energy×Freq Fig. 4(c) provides the results of our findings that are most striking: the number of joules required to execute the task of sorting has a sweet spot at 1.8 GHz. The energy consumption declines by approximately 25% (708 J) when the CPU frequency is changed from turbo mode to 1.8 GHz.

For the other algorithms, the results are also interesting. std::sort is more energy efficient at 2.4 GHz. This corresponds to energy savings of approximately 12% (or 176 J) compared to the turbo mode, which leads to the shortest response time. Running the CPU at a low frequency, i.e., 1.2 GHz, increases the energy consumed by 25% (427 J) compared to the sweet spot frequency. Radix sort is more energy-efficient at 2.2 GHz, which corresponds to energy savings of approximately 15% (or 96 J) compared to the turbo mode and 18% (119 J) compared to the lowest frequency.

Fig. 4 clearly shows the different sweet spot frequencies (vertical dotted line), in ascending order, for counting, radix and std::sort. The lines in the diagram correspond to list sizes of 10..50 million elements.

Table 3 provides an overview of the results and identifies the sweet spots for each algorithm (sweet spots are marked with a star '*'), and the energy savings that can be achieved when the most energy-efficient CPU clock frequency is selected compared to using the maximum frequency (i.e., turbo mode).

In order to gain more insights, we calculate the energy savings from running the algorithms at the sweet spot frequency compared to the maximum frequency

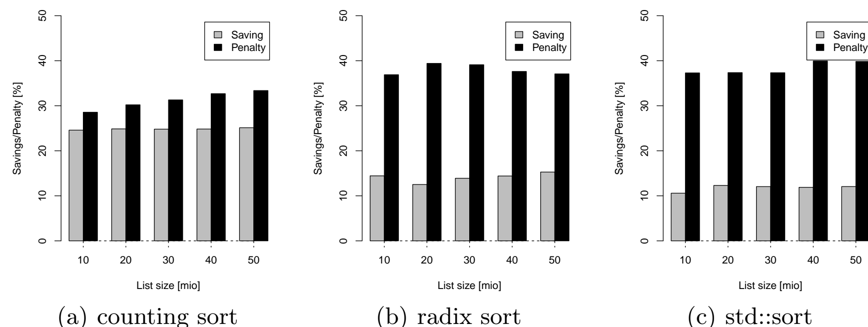


Fig. 5. AC energy saving compared with time penalty.

(AC energy saving) and the associated loss of performance (time penalty). The penalty is always higher than the savings. Fig. 5 compares AC energy savings and time penalties for all three algorithms and all list sizes.

While researchers have already found that energy-efficiency can be achieved by redesigning software code, by making better use of memory and, by using more efficient hardware components (see [13]), it is not well known that the energy-efficiency of software is also affected by the frequency of the CPU at very precise frequencies other than the maximum frequency. Namely, our findings provide an answer (A) to our two research questions:

- A2a (Effectiveness). Software energy efficiency can be improved by choosing the most adequate CPU clock frequency. CPU clock frequency leads to a considerable variability of the energy needed to complete a data-intensive computational task.
- A2b (Determinability). The CPU clock frequency which makes software more energy efficient can be determined. In the case of counting sort for 50mio elements, reducing the clock frequency by ≈60% of its maximal speed can lead to an energy reduction of 25%.

These results entail not only that for data analytics tasks the sweet spot of CPU frequency can, and should be determined, but it also shows that data-intensive tasks with the same algorithmic complexity can have a different energy

Table 3. The energy-efficient sweet spot of sorting algorithms.

Algorithm	E (J)	Freq. (MHz)	t (s)	P (W)
radix	530.8	*2200	2.6	204.2
radix	626.9	turbo	1.9	330.4
std::sort	1282.3	*2400	5.9	216.9
std::sort	1464.1	turbo	4.2	346.3
count.	2093.8	*1800	11.3	184.9
count.	2801.5	turbo	8.5	330.3

efficiency. Therefore, it seems natural to consider developing energy-efficiency benchmarks for software applications. While ISO software quality parameters include over 50 metrics [14], SPEC CPU2006 provides comparative studies on hardware performance and SPECpower for hardware energy efficiency[6], the same does not happen to software energy efficiency.

5 Dynamic Software Adaptation

Our idea to implement a new kind of self-adaptive software for data analytics and business intelligence is to enable data-intensive tasks to select the frequency of the CPU, based on the type of data processing. We will demonstrate how such a system could be implemented using the computing task evaluated: sorting. When a self-adaptive software for data analytics receives a request, it uses optimization techniques to determine at which frequency the CPU should be clocked to fulfill the constraints of the request: performance or the energy efficiency of the execution, or a combination of both. Optimization uses the approximated functions from Table 4 and the number of elements requested to be sorted. The information is stored in so-called QoS contracts. The result of the optimization is the CPU frequency at which the sorting algorithm should be executed. Thus, we propose a three-phase approach:

1. Approximate functions of sweet spot frequencies based on micro-benchmarks to determine a server's individual sweet spot frequency.
2. QoS contracts to capture the assessed non-functional behavior.
3. Optimization to compute the optimal frequency and algorithm for a given user request at runtime.

We extend previous work (see [15]) by incorporating hardware reconfiguration by means of explicit frequency scaling.

5.1 Approximate functions

It is possible to predict the sweet spot frequency by approximating a function of the energy consumption depending on the frequency using multiple linear regression and searching the minimum value of this function for a given list size. Fig. 4 depicts the energy consumption across all possible frequencies for sort invocations of the three investigated sort algorithms. For each algorithm a sweet spot frequency can be determined independently of the list size.

For the measurements of radix sort, fourth grade polynomial functions approximate the measured values very precisely as shown in Table 4. The first five rows show functions for 10..50 millions elements, whereas the last row represents the generic function $E(freq, size) = a \times freq + b \times freq^2 + c \times freq^3 + d \times freq^4 + e + f \times listsize$ with an adjusted R^2 of more then 99%. The minimum of this general function is at 2.4 GHz for all list sizes, which is not the measured

[6] https://www.spec.org

e (intercept)	$a \times freq$	$b \times freq^2$	$c \times freq^3$	$d \times freq^4$	$f \times listsize$	ars
98.558887	-17.000578	24.473429	1.497380	5.577048	1.00E+07 (fixed)	0.93990
214.645925	-30.237388	45.255654	2.977515	10.231982	2.00E+07 (fixed)	0.96998
344.115647	-43.960454	78.485585	1.588671	15.814538	3.00E+07 (fixed)	0.95190
450.068206	-49.751193	103.690842	0.926805	21.180903	4.00E+07 (fixed)	0.96550
564.112603	-51.629321	131.365326	6.015857	26.856475	5.00E+07 (fixed)	0.95546
-15.65866	-86.12392	171.4039	5.816562	35.62546	1.16653E-05	0.9942

Table 4. Fourth grade polynomial functions for radix sort on 10 to 50 million elements (ars = adj.r.squared).

mean sweet spot at 2.2 GHz, but is less than 1% distant from it. The cause of this difference is the (small) deviation of the approximated function and the closeness of the frequencies around the sweet spot frequency.

Thus, to automatically determine a sweet spot frequency on a target platform (unknown at design time), a developer has to provide a (micro) benchmark for different algorithmic implementations. Using the approach described above, the system can compute the sweet spot frequency automatically.

5.2 QoS Contracts

The Quality Contract Language (QCL) [15] allows to capture the non-functional behavior of an implementation. A contract in QCL specifies for an implementation of a data analytics task (e.g., radix sort) pairs of non-functional provisions and requirements. If the requirements are fulfilled, the provisions are guaranteed to hold. Listing 1.1 depicts an example of a QCL contract for the radix sort implementation used in this paper. It specifies 2 modes, which are alternative pairs of requirements and provisions. The two modes represent the most energy efficient and the fastest way to execute the algorithm. Thus, for the first mode, the sweet spot frequency determined in the previous phase is specified as a requirement on the CPU. The second mode specifies the highest possible frequency as a requirement. In addition, the runtime and energy consumption on the CPU are specified as functions depending on the list size. They are determined analogously to the sweet spot functions in the first phase. All modes specify which guarantees are given if a set of requirements is fulfilled. In the example contract, a specific maximum response time, which is equal to the time required on the CPU and a small overhead (`x1` and `x2`, respectively), is guaranteed.

```
1 contract Radixsort implements Sort.sort {
2   mode efficient {
3     requires resource CPU {
4       frequency = 2.400 [MHz]
5       max time = f1<list_size> [ms]
6       max energy = f2<list_size> [J] }
7     provides max response_time = time + x1
8   }
9   mode fastest {
```

```
10     requires resource CPU {
11       frequency = 3.000 [MHz]
12       max time = f3<list_size> [ms]
13       max energy = f4<list_size> [J] }
14     provides max response_time = time + x2
15   }
16 }
```

Listing 1.1. Example of a QCL contract for radix sort.

Contracts, like the discussed example, can then be used to generate problem formulations for off-the-shelf constraint satisfaction and optimization problem solvers.

5.3 Optimization

To determine the service with a particular CPU frequency an algorithm should be run on, we use an integer linear program (ILP) as shown in Listing 1.2. For clarity, we only show an ILP example for the decision whether radix sort shall be executed on one of two servers with different CPU frequencies in either the most energy-efficient or the fastest way. In the example, all values referring to server $N1$ correspond to the measurement values shown in the last section for a sort request of 30 million elements. The values referring to server $N2$ are not based on measurements, but introduced to show the general applicability of the approach to multiple servers.

```
1  min: energy#N1 + energy#N2;
2  //decide for one server and variant
3  b#rdx#eff#N1 + b#rdx#fast#N1 + b#rdx#eff#N2 + b#rdx#fast#N2
4  = 1;
5  //approximated runtime per decision
6  time#N1 =              1620b#rdx#eff#N1 + 1164b#rdx#fast#N1;
7  //base load + decision-induced consumption
8  energy#N1 =       97 +  324b#rdx#eff#N1 +  376b#rdx#fast#N1;
9  //min frequency + sweet spot-min frequency
10 frequency#N1 = 1200 + 1000b#rdx#eff#N1 + 2000b#rdx#fast#N1;
11 time#N2 =              1120b#rdx#eff#N2 +  940b#rdx#fast#N2;
12 energy#N2 =      100 +  420b#rdx#eff#N2 +  530b#rdx#fast#N2;
13 frequency#N2 = 1200 + 1000b#rdx#eff#N2 + 2000b#rdx#fast#N2;
14
15 bin b#rdx#eff#N1,b#rdx#fast#N1,b#rdx#eff#N2,b#rdx#fast#N2;
```

Listing 1.2. Example of an Integer Linear Program for self-adaptive software.

The ILP example comprises 4 decision variables and 3 usage variables per server. The decision variables of the optimization problem have the form:

$$b\#algorithm\#variant\#server$$

The prefixing b denotes the Boolean type of the variable, which is explicitly stated as constraint on lines 22 and 23, and *algorithm*, *variant* and *server*,

delimited by pounds (#), denote the respective algorithm, variant (energy efficient or fast) and server. The meaning of $b\#rdx\#eff\#N1 = 1$ is the decision to use radix sort in its energy-efficient variant on server $N1$. Thus, each decision is represented by a variable. Each server is characterized by the variables $time$, $energy$ and $frequency$, which include the name of the server separated by a pound. They denote the time required on, the energy spent by and the CPU frequency used for the respective server.

The objective function of the ILP is a linear combination of the problem's variables. In the example, the objective function specifies the goal to minimize the energy consumption of both servers for user requests to sort n elements.

Then, two types of constraints are generated for a specific request (e.g., the invocation of sort for a list of 30 million elements as in the example above). First, a structural constraint to ensure the selection of at least one variant is generated (cf. line 3 and 4). Second, for each server variable (time, energy and frequency) a constraint, reflecting the impact of a decision on them is generated. For example, the constraint in line 6 and 7 specifies that deciding for radix sort in efficient mode on server N1 will require 1620 ms, whereas using fast mode will require only 1164 ms. For energy consumption, the idle consumption has to be considered in addition. The constraint on line 9 and 10 reflects an idle consumption of 97 J and the respective consumption of radix sort in the most efficient and fastest mode. Considering the idle consumption is important if the servers are always powered. The frequency constraint on line 12 and 13 reflects the minimum possible frequency (1200 MHz), the sweet spot frequency (1000+1200 MHz), and the maximum possible frequency (1000+2000 MHz).

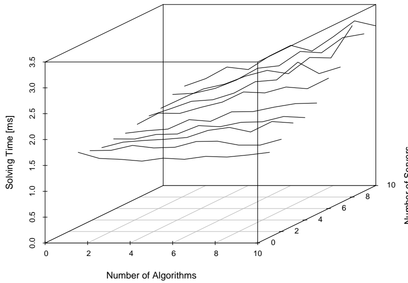


Fig. 6. Time to Compute Adaptation Decision.

To solve this optimization problem, standard solvers like LP Solve [16] can be applied. The time required to solve problems as shown in the example, depends on the number of modes (i.e., execution variants) for which we used $m = 2$ representing the most efficient and the fastest variant, the number of algorithms for the same task A (e.g., different sort algorithms) and the number of servers N. Fig. 6 shows the time required to solve ILPs with $A = [1..10]$ algorithms with 2 modes each on $N = [2..10]$ servers. As can be seen, the time to derive

the decision is negligible. It took ≈3.1 ms to identify which algorithm out of 10 to run on which of 10 servers.

5.4 Limitations

The approach we presented is feasible for Big Data intensive tasks since the processing of one single task can take several minutes. For tasks with a short duration (typically less that 2 seconds), managed by multi-process operating systems, the approach is generally not, or at least not easily, applicable. If two distinct data-intensive tasks have different sweet spot frequencies and the tasks are multi-tasked, the operating system needs to switch the sweet spot frequencies rapidly or choose the frequency that minimizes the overall energy consumption. Nonetheless, the future generation of processors will most likely enable setting the frequency of individual cores, making the sweet spot approach valid for a broader type of tasks beyond data-intensive ones.

6 Related Work and Future work

The closest existing research to our work was conducted by Livingston et al. [3]. Their work classifies software applications as memory- and compute-bound. For memory-bound applications, they demonstrate that a higher energy-efficiency is achieved at lower CPU frequencies since memory behaves as a bottleneck. For compute-bound applications, a higher energy-efficiency is achieved at higher CPU frequencies since finishing work quickly is the best approach for efficiency. For algorithms which cannot be purely classified as memory- or compute-bound, they propose to use sweet spot frequencies, a benchmarked optimal frequency between the lowest and highest frequency. Our work confirms the findings by Livingston et al. also in newer computer architectures and makes a detailed analysis of sweet spot frequencies.

Other related work can be classified taking into account the level at which energy-efficiency analysis was conducted. We use the terms macro-, meso-, and micro-level to express studies conducted with large software applications, algorithms, and instructions. At the macro-level, researchers (cf. [13]) have looked into the energy-efficiency of large management information systems such as ERP, CRM, and databases. While it is important to look into the efficiency of such systems to identify fields of improvements, the approach taken does not allow to gather insights on how software could be re-engineered differently to obtain energy reductions. At the meso-level, Bunse et al. [17] evaluate various sorting algorithms in battery powered mobile communication using smartphones. The results indicate that insertion sort is most efficient. Rivoire et al. [18] investigate system-level benchmarks for sorting. Nonetheless, the work does not explore the effect of CPU frequency on software energy-efficiency. At the micro-level, Ong and Yan [19] use an abstract machine to study the energy consumption of search and sorting algorithms. The energy requirement of each instruction was estimated and, e.g., an ALU access consumes 8×10^{-12} joule per 32 bits. Their

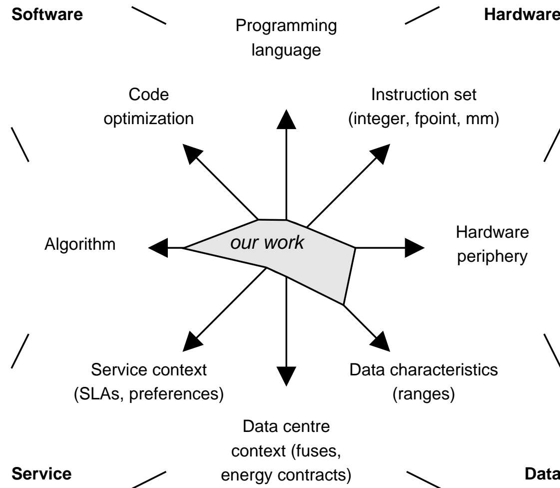


Fig. 7. Dimensions and directions of energy-efficient software research.

findings indicate that the energy consumption can differ in orders of magnitude between algorithms, and, also, that faster algorithms can sometimes consume more energy than slower ones. In [20], the authors propose a first-order, linear power estimation model that uses performance counters to estimate CPU and memory consumption. The accuracy of the model estimates consumption within 4% of the measured CPU consumption.

Our long-term research goal is to study how energy-efficient mechanisms can be implementation as part of self-adaptive software and service systems that change their behavior and implementation, and affect the computing environment to reduce energy consumption. Fig. 7 shows relevant dimensions of research in this field.

7 Conclusion

Software applications for data analytics over Big Data typically have a high energy consumption. In this paper we studied a mechanism to make data-intensive processing tasks more energy efficient. Our findings indicate that the existence of CPU sweet spots can be explored in three ways: 1) by adapting the CPU frequency to sweet spots, the maximum power used by data analytics tasks can be established; 2) the use of sweet spots leads to effective energy gains of up to 25% ; and 3) sweet spots enable the design of new and more efficient self-adaptive software architectures. Our results are important, especially for large cloud data centers (e.g., Facebook, LinkedIn, and Twitter), since they can lead to the design of new software and scheduling policies to reduce energy consumption of data analytics and business intelligence applications.

Acknowledgements

This work has been partially funded by the German Research Foundation (DFG) under project agreements SFB 912/1 2011 and SCHI 402/11-1.

This is an "executable" paper. All measurement results, executable source code used for the experiments, logs and traces can be found online at the experimental results platform Areca[7].

References

[1] J. Koomey., "Growth in Data center electricity use 2005 to 2010," Analytics Press, 2011. [Online]. Available: http://www.analyticspress.com/datacenters.html

[2] P. Zadrozny and R. Kodali, *Big Data Analytics Using Splunk: Deriving Operational Intelligence from Social Media, Machine Data, Existing Data Warehouses, and Other Real-Time Streaming Sources.* Apress, 2013. [Online]. Available: http://books.google.de/books?id=CK9AYF8WTsoC

[3] K. Livingston, N. Triquenaux, T. Fighiera, J. Beyler, and W. Jalby, "Computer using too much power? give it a rest (runtime energy saving technology)," *Computer Science - Research and Development*, pp. 1–8, 2012. [Online]. Available: http://dx.doi.org/10.1007/s00450-012-0226-0

[4] K. Choi, R. Soma, and M. Pedram, "Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Trade-Off Based on the Ratio of Off-Chip Access to On-Chip Computation Times," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*, February 2004, Paris, France.

[5] L. Brochard, R. Panda, and F. Thomas, "Power consumption of clusters: Control and Optimization," Industry Talk at Fourth International Conference on Energy-Aware High Performance Computing (EnA-HPC), September 2013.

[6] D. Versick, I. Waßmann, and D. Tavangarian, "Power consumption estimation of CPU and peripheral components in virtual machines," *ACM SIGAPP Applied Computing Review*, vol. 13, no. 3, pp. 17–25, September 2013.

[7] D. E. Knuth, *The Art of Computer Programming.* Addison-Wesley, 1998, vol. 3 - Sorting and Searching, 2nd Edition.

[8] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics." in *CIDR*, vol. 11, 2011, pp. 261–272.

[9] D. Hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, vol. 0, pp. 194–204, 2013.

[10] R. Schöne, D. Hackenberg, and D. Molka, "Memory performance at reduced cpu clock speeds: an analysis of current x86_64 processors," in *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, ser. HotPower'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387869.2387878

[7] `http://areca.co/26/The-Cost-of-Sorting`

[11] M. P. I. Forum. (2012, Sep.) Mpi: A message-passing interface standard - version 3.0. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[12] ISO/IEC, "ISO/IEC 14882:2011: Programming languages – C++," Tech. Rep., 1998.

[13] E. Capra, C. Francalanci, and S. Slaughter, "Measuring application software energy efficiency," *IT Professional*, vol. 14, no. 2, pp. 54–61, March 2012.

[14] ISO/IEC, "ISO/IEC 25010: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," Tech. Rep., 2010.

[15] S. Götz, C. Wilke, S. Richly, and U. Aßmann, "Approximating quality contracts for energy auto-tuning software," in *Proceedings of First International Workshop on Green and Sustainable Software (GREENS 2012)*, 2012.

[16] K. Eikland and P. Notebaert, "LP Solve 5.5 reference guide," http://lpsolve.sourceforge.net/5.5/ (access on 26.11.2012).

[17] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, "Choosing the best sorting algorithm for optimal energy consumption," in *Proceedings of the International Conference on Software and Data Technologies (ICSOFT)*, 2009, pp. 199–206.

[18] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis, "Joulesort: A balanced energy-efficiency benchmark," in *Proceedings of the ACM SIGMOD Intl. Conference on Management of Data (SIGMOD)*, 2007.

[19] P.-W. Ong and R.-H. Yan, "Power-conscious software design-a framework for modeling software on hardware," in *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, Oct 1994, pp. 36–37.

[20] G. Contreras and M. Martonosi, "Power prediction for intel xscale reg; processors using performance monitoring unit events," in *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on*, Aug 2005, pp. 221–226.